UiO : Department of Informatics
University of Oslo

Implementation of editors for text-based domain-specific languages

A ThingML use case

Kyrre Havik Eriksen
Master Thesis Spring 2013

Implementation of editors for text-based domain-specific languages

Kyrre Havik Eriksen

30th April 2013

Acknowledgements

I would like to express my special thanks of gratitude to my supervisor Franck Flurey, who gave me the golden opportunity for this master thesis on the topic of text-editors and domain-specific languages.

Secondly I would also like to thank my parents and friends (especially the 8th floor and Kristina Heyerdahl Elfving) for encouraging words and keeping me company.

I would also like to thank Bao Marianna Nguyen for the love and support, and believing that I would finish before the deadline.
Thank you

Kyrre Havik Eriksen

Abstract

Keywords: ThingML[1], Open-source, Framework, Editor, Integrated development environment (IDE), Domain-specific language (DSL), Eclipse, RsyntaxTextArea, Eclipse Modeling Framework, EMFText, Xtext.

In this thesis there has been a research on different text-editor frameworks that can be easily implemented with a domain-specific language. Several frameworks were tested both by implementing a language and by a comparison of available data. From this we ended up favouring RSyntaxTextArea and made a fully working editor with the language ThingML. Features implemented were syntax highlighting, code suggestion/completion, outline/syntax tree, and error-reporting. These selected features were chosen by having a survey of which gathered information on what kind of editors developers use, what kind of features the editors supports and what features that were used.

[1] http://thingml.org/

Contents

List of Figures

List of Tables

Listings

Part I

Introduction

Chapter 1

Introduction

Domain-specific languages (DSL) are becoming more and more popular and are widely used, from game-engines[1] to statistical modelling languages. A DSL is usually constructed with a specific domain or task in mind. A good example is HTML[2] for creating web pages and Logo which is an educational language where you paint with a turtle. When a domain-specific language is complete, it often needs an editor that understand the language. This is often solved by either using a framework that also provides an editor for the new language, such as the Eclipse Modeling Framework[3]. Another solution is to let the end-user choose their own tool-sets and just provide a command line access to the language's compiler. An advantage of this approach is that it is quick to test out the new language, and easy to distribute as there are fewer tool-sets to rely on. This is often the case with general-purpose languages. General-purpose languages are languages aimed at several different tasks and usually works on a wide range of systems, Java[4] is a typical example of a general-purpose language.

The problem with letting the users choose what tool to use, is that it makes it more complicated for the non-expert users. There is a wide range of tools to choose from, and it can be hard to decide what to use if you are not familiar with a tool from before. There can also be problems if there is one editor that is expected to be used, especially if this editor is generated by the tool that were used to design the domain-specific language[5]. Therefore this thesis focuses on simple text-editor frameworks, but still with enough features to be usable by expert users.

There are lots of articles discussing how to make domain-specific languages, and comparisons of different tools to use for this, but there is a lack of information on how to make editors for domain-specific languages. In the start of this thesis we discuss the different tools used for creating

[1] Like the Unreal Engine

[2] Technically not a programming language as it is not Turing complete, but rather a markup language

[3] A modeling framework and code generator for building tools and languages based on structured models

[4] Both an object-oriented and concurrent language

[5] For example the Eclipse Modeling Framework generates an Eclipse plug-in for your language

domain-specific languages and present the language that is going to be used as a test case in this thesis, ThingML, a modelling language for embedded and distributed systems designed by Sintef[6].

Since there exists a lot of different frameworks, we conducted an informal survey at the University of Oslo. The outcome represented surprisingly few different types of frameworks. Internet were then used to supplement with other frameworks so that ended up with a handful of frameworks that could be suitable for this projects. With these frameworks we did some testing, both on implementation with our language and with the framework's own demo version. We also looked into the user base of each tool and how regularly they were updated. Some of the frameworks were too complex to even be considered, while some were so simple that it would be to much work to implement a language and extra features.

After settling down with a few different frameworks ranging in complexity, from just text editing as a feature, to an "Integrated development environment"[7] (IDE) plug-in. We first had to do some research about how programmers interact with editors and what kind of features they use. This data was collected from three different sources, first from a survey about editors and what kind of features the editors have, second by observing a well known editor, Eclipse, in use. Third we used the usage statistics from both NetBeans and Eclipse[8] to see which of the features that were most used, compared with the statistics from out survey about editors and features.

Most of the feedback from the data gathered told us that the majority of developers use a small set of different editors, while there is a large subset of simpler editors with few users. It's also similar with the feature usage, the majority use a hand-full of the features supported by the different tools. Which led us to focus on six different features to implement in the new editor. This was both to avoid making it too complex and to support only what was necessary at first.

After researching and testing different frameworks we ended up with one specific Java framework, which is already used in an IDE, but has been separated out to make it easier to implement it in other projects. The next phase was to implement our language, ThingML, with the new framework to see if it was possible, and what the necessary steps were. Nearly all of the chosen features were implemented, mostly because the framework laid the groundwork with templates for most of the features we wanted to use.

To see if the new editor was usable we conducted an observation with some of the developers familiar with the ThingML language. The focus on this observation was to find out how the editor compared to other editors and how users interact with it. The overall feedback was positive and that it reminded them of other editors, which in turn will make it easier for developers not familiar with the editor to start using it. Still there was some challenges and features that were overlooked.

[6] in Oslo, Norway.

[7] An integrated platform supporting more then just editing of code, often features such as debugging, file management, project management, and more

[8] Both Java specific IDE, with large user groups

1.1 Motivation

The motivation behind this master thesis present to find a suitable editor framework to implement domain-specific languages. When reading articles about creating domain-specific languages a lot of them end up making plug-ins for Eclipse or in worst case don't supplement a tool to work with at all. Both of these cases can be discouraging when you are first starting to work with a language. So what we wanted to do was to either create or find a framework suitable to be implemented with either a domain-specific language or a Eclipse model. So what we where looking for was a framework with the features as an IDE, but still be lightweight, similar to regular text editors.

Of course there are a lot of advantages in using Eclipse[9], but mainly these positive advantages only target developers already familiar with the Eclipse environment. This user group is typical Java and Scala developers. The challenge is the rest, which are not used to work with Eclipse. Either new developers learning their first language, or experienced developers wanting to try something new. Another challenge and something that can be deterrent with Eclipse is the file size of the downloaded file and the complex interface. Both of which can be intimidating for new developers and is also one of the challenges we want to address when looking for a new framework. It should be easy and encouraging to try a new language.

Based on this we are motivated to see if there are text-editor frameworks that help us in this process of making a new editor, or if needed, to start from scratch. It is also a goal to make it easy for other domain-specific language developers to implement their languages with the potential framework or editor. This is to help other to avoid the use of the Eclipse workbench when it is not needed, especially when targeting developer not familiar with the use of Eclipse, and to avoid the need to set time off to learn the end users how to utilise Eclipse.

1.2 Chapter Overview

This thesis is structured in four main parts. The background and state of the art on how implementation of editors for text-based domain-specific languages. There is also a section about the different research methods used in this thesis. Before this different tools that may be suitable for implementation of a domain-specific language are presented. Following with uses cases of implementation of a simple version of ThingML with a selections of the editors.

In the second part what have been done in the case of research is presented. It starts of with an survey of editors and their features, followed up by an observation of developers on different levels working with Eclipse. After this the different features which are going to be implemented in the new editor are presented and defended. This is also summing up the second part.

[9] A popular IDE

The third part of this thesis will present the implementation and developing of the editor with the case language ThingML, followed by a usability testing of the editor with developers familiar with ThingML.

In the final part a discussion on what have been done is brought up and future work that are needed to be done before the editor can be released as a final product.

Chapter 2

Research methods

In this section a discussion about the design process that has been a part of this thesis and the different research methods used are presented.

2.1 What do people want

When making something it is always important to ask what the user group want and need. In some cases this can be a difficult task, *especially* when making something new. In this case there are nothing new that has been made, but rather a different take on it. This also differentiate this thesis from other design project as the goal of the editor is not to create something new, but rather strip down something old. A good quote describing the design process in a whole is from Bratteteig & Stolterman[1];

> Design can be understood as a process that includes activities concerned with three levels of abstraction. At the most level we find a vision, at a more concrete level an operative image, and at the most concrete level we find the final design specification.

The goal of the editor is not to end up at as the final design, but something between the vision and a more concrete level. Based on this the focus of the observation where of developers using the Eclipse workbench, to see what kind of features they used, and to see how this can be used in a more stripped down editor as the ThingML editor. Using what was learned from the first observation, there was created certain tasks for a potential user to conduct using the new ThingML editor, and then again it was observed how the user interacted with the new editor. After both of the scenarios where done there was a comparison of the two.

If this project wanted to go to the next level (to the finalisation of the design specification), there should been arranged a participatory design workshop where the participants would first perform certain tasks similar to the observation using the editor, and then afterwards discuss in groups what they had learned and how the new editor can improve on this.

When doing research it is important to remember that it is only through reproduction we can trust the results from empirical research, and two that

“empirical research never produce certain knowledge”. It is also important to be aware of the research methods flaws, and it that it may be a good choice to mix methods or replicate research to overcome the flaws[3].

2.2 Survey

To find out what kind of editors users might want, a survey was conducted to find out what kind of features already represented in editors, and how much each features was used by developers with different levels of programming experience. Using a survey as a research method is a good way to get a large number of responses quickly from a population of users that is geographically dispersed[10].

2.3 Observation

Based on the survey, there was an observation on the use of Eclipse and the interaction with its features. There are discussions on using video while observing, but it where opt-outed of this since the goal of the observation was to see what major features the participants used. Another thing to consider with video recording is also the time it would take to analyse the potential videos, as it would probably be more beneficial spent to use on further test-subjects and observations[13]. Even though in older observations there were no possibility to record the observation, there were still the focus or emphasis to plan what is necessary. In other words, what can be ignored or what is noted/registered during the observation[8]. The article advocate the needs of good preparation in both mechanics of recording, definition of units of behaviour and the scope of observation. Under the observation, thinking aloud was also encouraged. Thinking out loud is used to help verbalise the action performed by the participant. Also one of the major benefits with thinking aloud or describing the action, is that the participant often subconsciously add comments on what they like or not.

2.4 Case studies

In the end, after the survey and observation, multiple case studies where carried out. The reason for choosing case study as a research method is the number of users now working with ThingML is low. In this setting, using case study as a research tool can be a useful for gathering requirements and evaluating the interface[10]. Of course when working with such a small sample group it is important to try not to generalise to much and to focus on how the use cases can be representative for the end user.

Chapter 3

Background and motivating example

This chapter presents some background information about the creation of a domain-specific-language (DSL) and discuss what has been done earlier with implementation of DSL's with different frameworks. At the end of this chapter the case for this thesis, the ThingML project is also presented.

3.1 Introduction to ThingML — A language and an editor

This master thesis is a part of a bigger project at the Networked Systems and Services department of SINTEF in Oslo, Norway. The goal of the project is to design a new domain-specific language called ThingML. ThingML is a modeling language for embedded and distributed systems. The name ThingML is a reference to the term *Internet of Things*[1] and stands for 'Thing' Modeling Language.

> The idea of ThingML is to develop a practical model-driven software engineering tool-chain which targets resource constrained embedded systems such as low-power sensor and microcontroller based devices. ThingML is developed as a domain-specific modeling language which includes concepts to describe both software components and communication protocols. The formalism used is a combination of architecture models, state machines and an imperative action language.[2]

In this thesis the focus will be on the tool set for ThingML. Up to now the tool set consist of two types of editors both letting the user create and edit ThingML code, transform ThingML models to diagrams, and a code generator that compile ThingML to different languages; examples are C, Java, Arduino, or Scala. The two editors we have are Eclipse

[1] First used by Kevin Ashton in 1999 see RFID Journal, 22 July 2009. Abgerufen am 8 April 2011 http://www.rfidjournal.com/article/view/4986

[2] http://www.thingml.org/

with the ThingML plug-in, and one made in Java and Swing based on a framework called JSyntaxPane. The heavyweight Eclipse plug-in is a good representation of what ThingML can offer, but the more simpler editor based on JSyntaxPane is really not a good substitute for Eclipse mostly because it does not support the features that a good editor should have. This necessarily doesn't have to be such a problem, and you can say that in some way there is only one proper editor supporting ThingML, namely the Eclipse plug-in.

3.1.1 The ThingML Eclipse plug-in

The problem with Eclipse is how it is built and how it may not be the best tool for a novice developer learning an additional language, or his or her first language[2].

Let us focus on the weight problem of Eclipse first. When talking about weight in this context I am not referring to the weight in a physical space, but how it handles with the computer and byte size. For example, the smallest version of Eclipse is "Eclipse for Testers"[3] and is only made for JUnit testing, the smallest Eclipse in byte size for Java developers is "Eclipse IDE for Java Developers" and is 150 megabyte[4]. The top most downloaded version of Eclipse is "Eclipse IDE for Java EE Developer" and has a size of 227 mb[5]. Compared to our JSyntaxPane editor which is only 3,9 mb, the difference is huge. Of course this is without the ThingML samples. With the Samples it is clocking in at 6,7 mb all together. So compared to the commonly downloaded Eclipse IDE the ThingML editor is nearly 33 times smaller. These are all just numbers, but when a developer is going to try a new language and a new tool the ease of use plays a big part. And a program with a small byte size is more likely to be downloaded then a couple of hundred megabytes. Of course developers already using Eclipse (typically Java and Android developers) would probably download the ThingML plug-in, but the thing is that ThingML is not necessarily targeted against experienced developers, but also for novice programmers, who maybe just have started coding.

Eclipse is also not targeted towards novice users with their complex navigation in the UI. One can say that the weight of the Eclipse UI is heavy, and with all the other features it holds it can be difficult to navigate to the few features that are necessary when developing with ThingML.

Another problem with Eclipse is how it feels to work with it. Eclipse is written in Java and it consists of an Eclipse core with several 'layers' of plug-ins. This makes it slow to start and sometimes unresponsive. This is also something we would like to avoid with a standalone editor, and something we manage with the current standalone editor. In this case a simple to use, and lightweight editor is an advantage and a selling point.

[3]For Linux as of April 11. 2013 the size is 95 MB.
[4]For Linux as of April 11. 2013.
[5]As of April 11. 2013 it has been downloaded 1,061,698 times

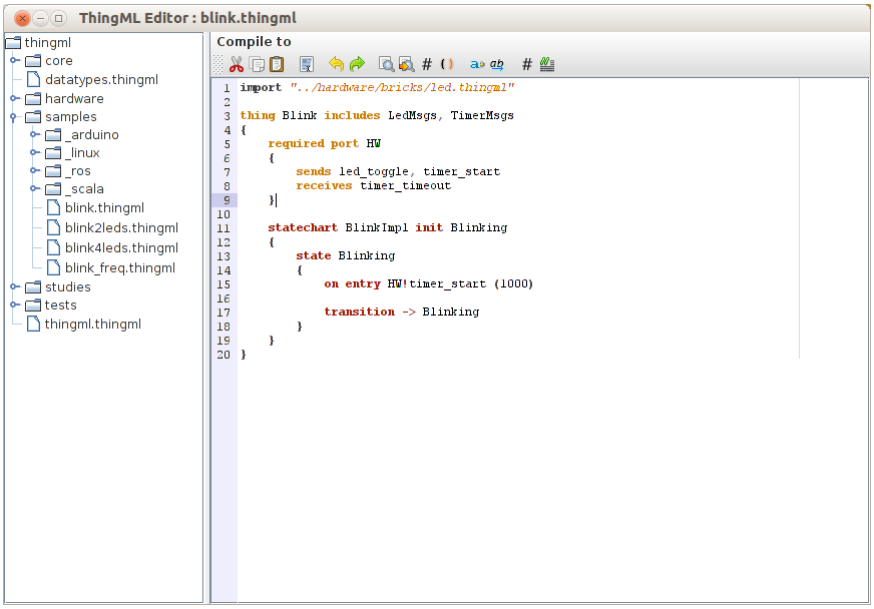


Figure 3.1: The old ThingML Editor using the JSyntaxPane framework.

3.1.2 The ThingML Editor

As mention above the current ThingML editor does not really represent the language very well. Mostly because it miss simple features like save and create file, or in other word project management. What happens now in the current editor is that every time the user would like to create a new file the user need to shut down the editor, use a file manager and create a file inside the ThingML project directory. First after following these steps, the user can open up the editor again and start writing in the newly created file. This is only a minor error and is not that big of a concern. The main problem with the editor is the lack of understanding of the ThingML language. This is typical for an editor, and is also probably why the JSyntaxPane framework only support syntax highlighting and code completion.

Figure 3.2 shows how the code completion work in the editor. When you start writing the framework looks for words using the same letters. There is no comprehension of *thing*, *statechart*, or *state* in the contex. Even though the only viable option is to refer the 'init' to the state 'Blinking' (as seen in line 13).

Another feature missing is something to give the developer an overview of the code. This is usually solved by having an outline or source tree of the code. This will help when working on bigger ThingML project. One thing that could be very beneficial by having an outline is to also represent the configuration file at the top of the tree. This is to ease the transition between the 'thing' and 'configuration'. Now if a developer want to compile the ThingML code, he or she needs to switch to the configuration window before compiling. Not a big issue, but a common mistake when first starting to work with ThingML.

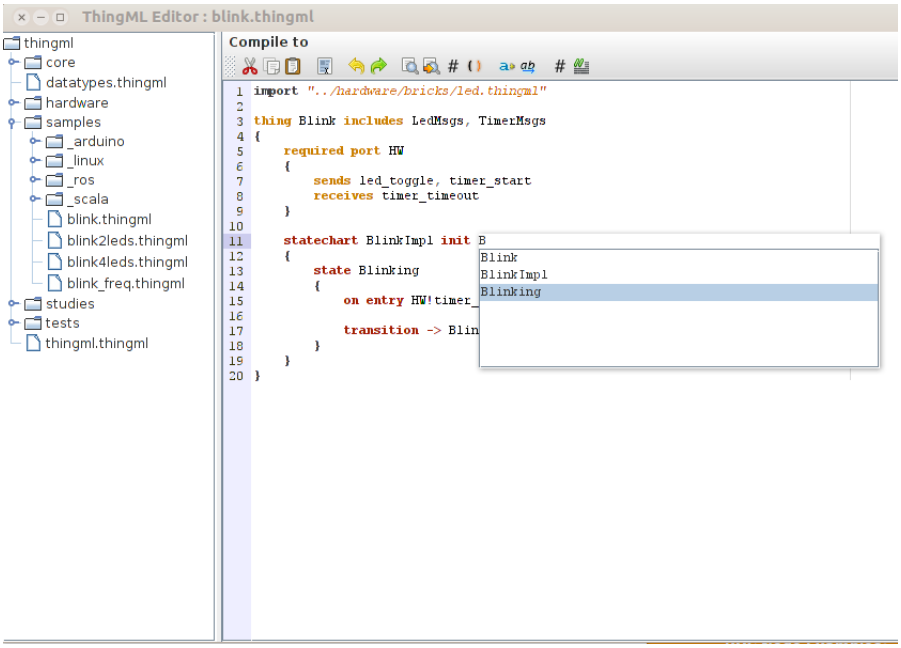


Figure 3.2: Code completion in the old ThingML editor.

One thing that the edit has and is doing rather well is the folder tree. This can be seen on the left in figure 3.1 and 3.2. It is not project based, but rather the ThingML folder you choose when starting the editor[6]. This is one of the features we need to transfer to the new editor.

There is also a small bug with the file management in the current editor, because it has problems reading files containing non-ASCI-letters[7].

3.2 The thesis question

This thesis question is based on the behavior of the two editors for the ThingML language, and the thesis question(s) is as follows:

> Is it possible to make a lightweight editor for a domain-specific language[8] without making it so complex and heavy-weight as an IDE[9]? And are there frameworks or editor already doing this job and thus answering this question?

This master thesis is focused on trying to find out if there are some frameworks that supports the thesis question(s).

[6]When starting the ThingML editor, it asks for the ThingML samples folder. This to supplement for the lack of file management mention earlier.

[7]Like æ, ø and å.

[8]Not necessarily only for a DSL

[9]Such as Eclipse

3.2.1 Why avoid a complex interface?

This is a rather good question and stems from how the user interact with the tool/workbench. Eclipse is a somewhat feature rich tool having solutions to most of the common and uncommon problems encountered while developing. It also helps the developer to speed up developing time. This is done by different plug-ins or extensions that are implemented in the editor, which of course are all editable. The problems with all these extra features are that the user needs to learn how to best utilize the different tools/plug-ins. This is a manageable task for an experienced user already familiar with the features of an IDE, but an inexperienced user (novice in other words) may and probably will struggle the first times using an IDE such as Eclipse. Therefore in the beginning the user will have problems using the features that the given tool support. Which is why avoiding a complex interface can be the smart solution. An introduction to a new tool (language in our case) should rather focus on what the user really needs and not what the tool really can do, and then represent that in an easy and understandable way. A good example is the idea behind the early Apple mouse where there was only one button for the user to click on. Since Steve Job meant that the user should not need the extra choice or be disturbed by them while working. This is often refereed to as a single button control interface[10]. Other examples of simplicity and editors with a non-complex interface are the Processing and Arduino tool, that focus only on the representative language. As editors they are pretty weak, but they give the novice user an easy start by not giving them much options other than; Writing code, compiling code, and look at examples. As a novice that is mostly what you need. A different example is teaching, as most teaching focus on the important part and try to describe or teach them in a simplified form. Then later when the student has gotten a broader understanding of the problem, more detailed information is given.

3.3 Simple ThingML

ThingML is a rather large and complex language, so in this thesis there will be used a simplified version called Simple ThingML in the examples and implementations. Simple ThingML is a sub set of ThingML, only featuring weighted transitions between states. The keywords used are as following:

StateMachine The state machine is the top level of the Simple ThingML code. In the real ThingML it is called *StateChart*.

State States are used to represent what kind of a state or behavior a StateMachine can be in. They are used to communicate and execute commands.

Transition Each state uses transitions to communicate and transfer

[10] Apple later relented on the usage of on button mouse, and in 1997 started support of mice with right-clicking

data between the states. To make the Simple ThingML a bit more advanced the transitions can be weighted[11].

Init The keyword 'init' is to note which state is the initial state in the state machine. There can be only one initializing state

Final The 'final' keyword is the opposite of the 'init', and represent which state is the finalizing state in the state machine. There can of course be several final states.

Listing 3.1 represent an example of the Simple ThingML language. The code shows three states, where s1 is the init-state, s3 is the final-state. Each of the three states have weighted transition to communicate with each other.

Listing 3.1: An example of a simple state machine with three states.

```
StateMachine statemachine {
  init s1
  final s3
  State s1 {
    -> (2) s2
    -> (1) s3
  }
  State s2 {
    -> (0) s2
    -> (5) s3
  }
  State s3 {
    -> (10) s2
  }
}
```

3.4 Related work

When starting of with this thesis a lot of the time was used to research on how other may have had implemented domain-specific languages with editors or workbenches. Unfortunately there weren't much litterateur covering this area. Most of the articles where either focusing on modeling tools to develop new domain-specific languages[6] or they compare different frameworks for the languages[14].

The article "Text-Based modeling"[6] discuss the the advantages and disadvantages about graphical versus textual modeling. What they argue for is that text-based modeling is a noteworthy alternative to the graphical ones [12]. This is based on advantages such as version control, speed of creation and formatting, and platform and tool independency. To help

[11] Weighted transition is not part of the real ThingML
[12] Page 6 in "Text-Based Modeling"

them in their case they have developed a framework called 'MontiCore'[13] which of course support text-based modeling.

The MontiCore framework is brought up in another article called "Efficient Editor Generation for Compositional DSLs in Eclipse"[9], which discuss the advantages of using Eclipse Modeling Framework for language and editor generation. MontiCore' advantage is that it is text-based which makes it an efficient tool for developing domain-specific languages. That is also why they are promoting textual modeling as an alternative to graphical modeling.

Another article not really discussion textual versus graphical is "A Comparison of Tool Support for Textual Domain-Specific Languages"[14]. Here they compare different textual tools for creating domain-specific languages. The following tools are tested; 'openArchitectureWare'[4], "Meta Programming System", 'MontiCore', "IDE Meta-Tooling Platform", "Textual Concrete Syntax", "Textual Editing Framework", and 'CodeWorker'. Each tool represent the concrete syntax as textual and they all allow for generation of text editors, ranging from simple text editors to code completion and validation while typing. There is no clear winner of which tool is best to use, but they point out that "Meta Programming System" is the only tool that don't generate a workbench based on Eclipse, but rather uses a cell based editor instead of free text.

When working with text-based domain-specific languages there is always a choice of which tool to use, especially for the parser. EMFText and Xtext are two big framework for building abstract syntax tree and parser. Both uses EMF as their base and they both produce a highly customizable Eclipse-based IDE. The article "Create a DSL in Eclipse - Open Source Tools to create DSLs"[7] heavily focus on why you should use EMF when creating a domain-specific language, but it also have an interesting discussion on whether to go for Xtext or EMFText. The author prefer Xtext over EMFText, but have no empirical data to back it up. Another article promoting Xtext is "oAW xText: A framework for textual DSLs"[4], which was written in the early days of Xtext. Some of the shortcomings brought up in this article is now fixed. The article "Textual Modeling Tools: Overview and Comparison of Language Workbenches"[11] features yet another comparison with Xtext and EMFText, but the focus here is rather on text-based modeling or textual projection. There was no real comparison between EMFText and Xtext, but the conclusion points out that "the combination of modular languages, different DSLs combined is much easier with projectional editors"[11]. It is also important to note that the texutal language workbenches where all Eclipse. Another article discussing the different frameworks to use is "Classification of Concrete Textual Syntax Mapping Approaches"[5]. They argument for the use of textual concrete syntax when modeling as it foster usability and productivity. They do a systematic analysis on creation of concrete textual syntax using a schema to map features in different tools. In their conclusion they identify problems with these framework when used in large-scale enterprise model driven

[13] http://monticore.org/

software development environment. They end the paper stating that the contribution is useful for third parties to choose a framework based on the presented classifications.

3.5 Textual versus graphical frameworks

When we talk about creating a domain-specific language there are different decisions to take. One is to choose to start with the abstract syntax[14] and then later decide how the language should be reflected in the concrete syntax. Then there is the other way around, where you create the concrete syntax[15] first, and then derive the abstract syntax from the concrete one.

In the later years it has become more popular to use graphical frameworks to model the syntax. Compared to the more traditional way of using a textual framework where you code or write the syntax. Of course there are also frameworks letting you use a mix of both worlds.

When choosing between textual or graphical framework, the language that is going to be built plays a big role. This mostly depends on if you want to start with the concrete syntax or the abstract syntax.

3.5.1 Eclipse Modeling Framework a graphical framework

Figure 3.3 presents an example of a meta-model created using Eclipse Modeling Framework (EMF). Based on this Ecore Model, EMF will generate (in it simplest form) a set of tools for your language, including a set of Java classes, adapter classes for viewing and command-based editing of the model, and a basic editor (an Eclipse workbench/plug-in). The advantages of having a graphical meta-model[16] is that it lets non-developers get a clear and easy-to-understand overview of the domain-specific-language. It is also easier to take a quick glance over a new language when every class and attributes are represented with images, as long as the image isn't too complex, preferably on one page[6]. This is partly because the scalability of a graphical meta-model isn't that high, and when the model get to complex the readability will decrease. The disadvantages of having graphical models is the problem of sharing them. Most tools do not support implementing smaller images or models, but force you do draw them all over again. Still, the biggest drawback is the lack of open standards that let the users decide which tools he/she want to use. Even thought Eclipse Modeling Framework uses an open XML-standard for their Ecore Models that other can, if they want, implement.

[14] The abstract syntax is the set of trees used to represent trees in the implementation

[15] The concrete syntax is defined by the context free grammar.

[16] An abstract syntax to be more precise.

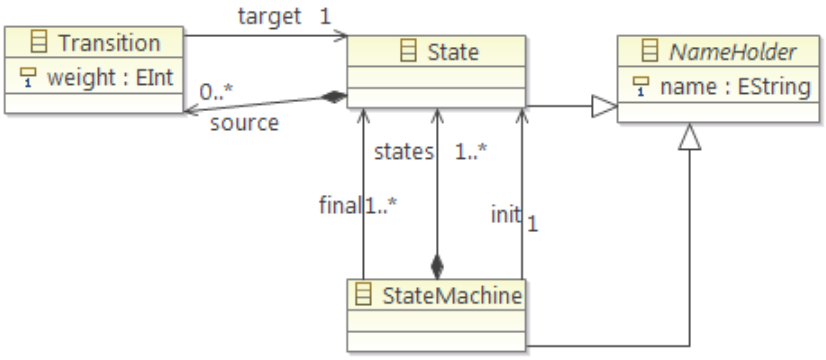


Figure 3.3: Simple state machine meta-model represented in Eclipse Modeling Framework

3.5.2 Textual Modeling

Textual modeling on the other hand has many of the disadvantages from the graphical modeling as their advantages. The developer can use any tool set they like, and with a distributed version control system, (like SVN or Git) sharing is easy. Still, they are confined to one parser or code generator, as most parsers or code generators don't mix.

Getting input or advices from other team members is also simple, since inputting a line or two of code into a text document will in no way force developer to redraw the model or modify it in any other way than pure copy and paste.

3.5.3 Working with a model

Another advantage with textual modeling is that writing a model is a lot faster than drawing one for a common developer, as you can focus on using the tool you like with a keyboard, in contrast to a graphical one where you need to use both a mouse and keyboard.

The following quotation describes a regular graphical modeling work scenario.

> First you need to use your mouse to drag the chosen element into your model-view, then you need to right click the model and choose properties (or the properties window is integrated as a side-bar and you only have to click the element) before you can start clicking the areas you want to fill in. Often the properties view consist of several tabs fitting different options.

Using a graphical tool to model is a lot more time consuming compared to writing. Especially for a developer who are used to typing. The drawback by having it all in text is that getting an overview of the created model can be difficult, even for an experienced developer. Luckily this can be solved by

generating a graphical view of the model, as this is a lot easier than doing it the other way around. Other challenges with textual modeling is discussed in Classification of Concrete Textual Syntax Mapping Approaches[5], where some of the main points are incremental parsing and model updating.

3.6 Summary

In this chapter the difference between textual and graphical DSL frameworks has been presented, some about EMF, textual modeling, and how it is to work with a DSL-model. The thesis case was introduced, with a simplified version of ThingML, "Simple ThingML" which is used for testing, and a bit about the current editor for ThingML and the problems or challenges with it.

When starting making a new domain-specific language it is always a decision to be made about how to develop the new language. This thesis is not about which solution is best, but both parties where presented in this chapter to give readers a reference point. The ThingML project where done using Eclipse Modeling Framework and EMFtext[17]. This was mostly done as a preference from the development theme and what they already knew. The perk of using EMF as a DSL tool is that you quickly get an Eclipse workbench to test out your language.

There was also a discussion about what it means for an editor to be lightweight, and how that may be important for developers using ThingML, since they are more likely to want to use a lightweight editor on a couple of mb instead of Eclipse which can be over 200 mb to download. We feel that having a good lightweight editor will be a huge advantage point considering the novice users.

[17] Read more about EMFtext in chapter 5

Chapter 4

State of the art

This chapter discusses if anyone has made the type of lightweight editor with the IDE features that we think is necessary. In the beginning there is a description of projects that have done similar work, and some that have focused on how to target new users and their usability needs. The chapter continues on with 11 tools that were picked out for further analyzing, after which six tools were chosen to make a working prototype using Simple ThingML as example language.

4.1 Processing Development Environment

A good example of a project focusing on a lightweight editor simple for beginners to use is Processing[1]. Figure 4.1 show how the tool looks like in Ubuntu. Processing is built on the Processing Development Environment (PDE), and the framework is increasing in popularity and several other project have adapted the framework for their own software[2].

> Processing is an open source programming language and environment for people who want to create images, animations, and interactions.

The Processing language is built upon Java, but uses a simpler language and syntax. This is to ease the work for the developer wishing to prototype an idea. The programming environment is also following this philosophy with few features at the developers disposal. The only automatic feature that the environment supports is syntax highlighting. If you want outline for example you need to download a third party plug-in.

There are no form for error-reporting when coding. The only error-reporting supported by the PDE is compiler-errors. Since it is using the Java Compiler for compiling, which is not that bad per se. The problem with this is the way Processing handles classes, because it wraps its own class around all the other classes that you may have in your project. This makes the information given back to Processing, and in turn to the programmer,

[1] http://processing.org

[2] Projects like Arduino and Fritzing are both buildt on PDE. See http://arduino.cc/ and http://fritzing.org/

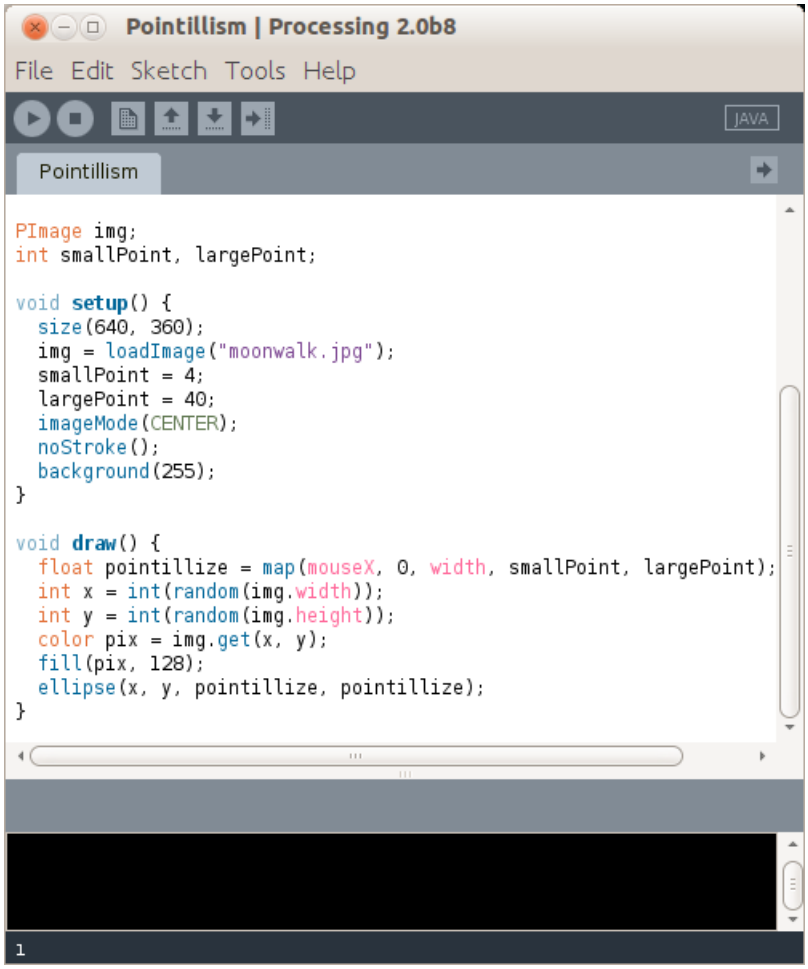


Figure 4.1: The Processing Development Environment

imprecise. In other words, there is only one file that gets sent to the Java Compiler, while in the PDE there are up to several. This is luckily worst case, and they do some minor pre-processing of the Processing code before it is compiled. This pre-processing do know where a certain syntax error is.

4.1.1 Could it be used?

When we started working with this thesis we were curious if it would be possible to use the Processing Development Environment as a framework. It is possible, but it doesn't look like the source code is meant to be used by others. The PDE framework code is really cluttered. The source-code for the Arduino editor is a good example, as some class names and methods are still using their Processing name. Even if Processing was a likely choice as our framework, there first needs to be a clean up of the code-base and then one must add the other features by hand. This means the PDE is not really an option.

The advantages of Processing is of course the lightweight software,

simplicity, and that it lets the user focus on writing code without the cluttering with creating project, and the process of project managing.

In one way you can say that what we want to achieve with our framework, is what Processing has managed with their Processing Development Environment.

4.2 Potential frameworks

There is a lot of different frameworks made for making text editors or IDE's. To try to narrow it down I conducted a small informal survey to gather information about what kind of frameworks and editors were used by students and professors at the University of Oslo. Unfortunately there wasn't much diversity in the use of editors or frameworks, and when further questioning about frameworks they had heard of, there wasn't much more response either. The only noteworthy editor frameworks were jEdit and Notepad++. As most of them answered either Emacs or Vi, or Eclipse or Netbeans.

So next step was to use the Internet to find frameworks that could be a suitable option. The table 12.1 in attachment 12.2 show just a sub set of either an editor with an open source framework, or just a plain framework. Both Eclipse, Emacs, Notepad++ and jEdit which was recommended from professors and fellow students are listed.

Things to notice is that a common denominator between all the editors is that they all support syntax highlighting, and some even only support that. Nearly half of the editors and frameworks support all of the features that we wanted a editor to have; syntax highlighting, code completion, source tree, and syntax folding.

The majority of the editors are also written in Java, and this is fortunate since we want the editor easily running on most platforms and most of the developers are also familiar with the Java programming language.

4.3 A closer look on some of the tools

In this section four of the chosen 11 frameworks are used to implement the Simple ThingML and see how they work. The four were picked out based on usability, familiarity, and if they are still updated. The chosen ones are Eclipse Modeling Framework together with EMFText, JSyntaxPane, JEdit, and RSyntaxTextArea. EMF and JSyntaxPane were chosen to see if it was possible to build on what we already have to see if it was possible to make Eclipse more lightweight and JSyntaxPane more rich on features.

4.3.1 Eclipse Modeling Framework and EMFText

Eclipse Modeling Framework is a powerful tool to make meta-models and parsers. EMFText is a plug-in for Eclipse that let you use HUTN-syntax[3] to

[3]Human-Usable Textual Notation — http://www.omg.org/spec/HUTN/

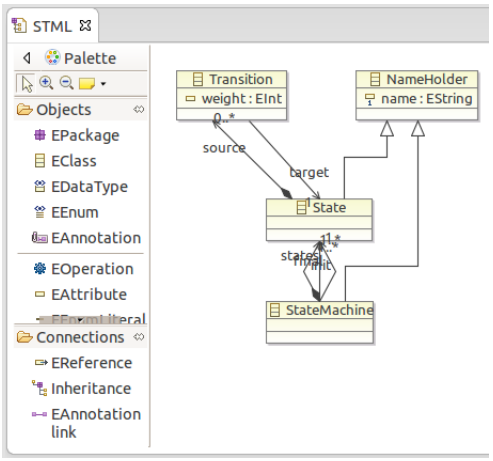


Figure 4.2: Ecore diagram of Simple ThingML

describe your language's syntax. This is all based on a Ecore digram created by Eclipse Modeling Framework. In figure 4.2 can you see the Ecore digram of an implementation of Simple ThingML. In listing 3.1 on page 14 you can see an example language described using EMFText. The EMTText code is a little more complicated and is linked to the Ecore digram. An example *rule* can be seen in listing 4.1. More about how Eclipse Modeling Framework and EMFText work in chapter 9

Listing 4.1: The Concrete Syntax Specification Language for Simple ThingML

```
RULES {
  StateMachine ::= "StateMachine" name[] "{" ("
     init" init[] | states | "final" final[])*
     "}";
  State ::= "State" name[] "{" ("->" source)*
     "}";
  Transition ::= "(" weight[] ")" target[];
}
```

Eclipse as a workbench is a powerful tool supporting all of the features we want in an editor naively. This is mainly done by by keeping a meta-model of the code written in its memory. When using Eclipse the only feature we then need to implement is the ThingML compile and code generator. The problem with Eclipse as a framework and workbench is the major size and memory hog it is, if you don't have a powerful computer, it will use a lot of time to start. This can often also be reinforced by adding other plug-ins to get more features. For example if you need to code in Scala. One way Eclipse has solved this is to have own Eclipse versions for different frameworks. A good example is the Eclipse Android Development Tool (Eclipse ADT) which is a standalone and semi-stripped down version of Eclipse supporting Android development. This feature was introduced in 'Juno', the latest Eclipse version released summer of 2012.

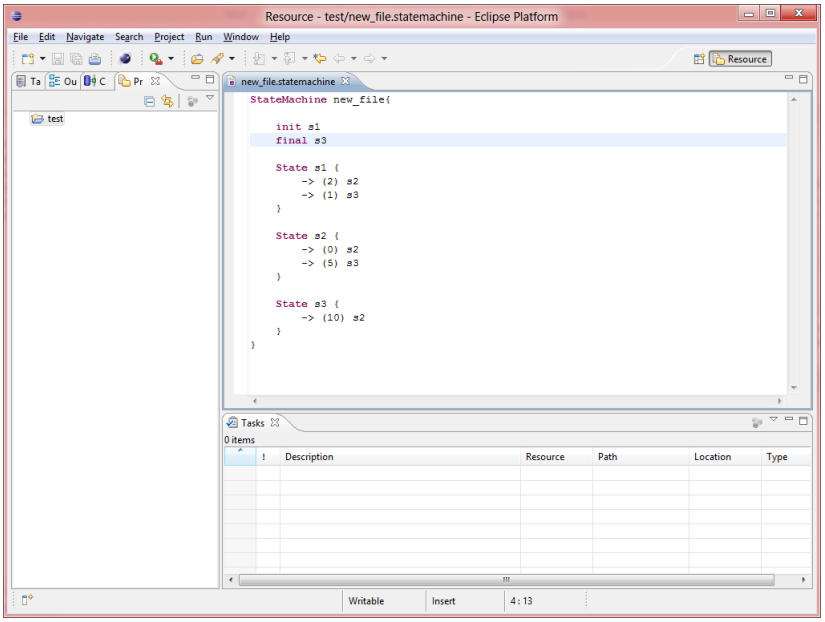


Figure 4.3: ThingML using Eclipse workbench

To sum it up; Eclipse is a great workbench with a lot of features supporting a wide variety of languages and is familiar IDE for developers. The drawback is that it is heavy to run, download, and complicated to use.

4.3.2 JSyntaxPane

JSyntaxPane[4] is the framework the initial ThingML editor is built on. The main goal of JSyntaxPane is to give the user a good looking Java Swing editor with syntax highlighting. This is also the main feature of the framework.

Figure 4.4 shows the implementation of Simple ThingML and a state machine. What you see there is the standard implmenetation of JSyntaxPane. As you can see there is not much going on in terms of features and toolbars.

On the plus side it is fairly simple to implement. All you need is a small .jar library, a Java class, and a jFlex[5] file for your language to get your editor up and running. The jflex is used to generate a lexer with all the keywords in the chosen language. This is used both for syntax highlighting and code suggestion. The framework has no understanding of the code written, so if you want error-reporting or content assist they have to be manually implemented. Still, going from code suggestion till content assist isn't that far if you already have a parser for your language.

The way it is structured now is that it is something someone would mostly want if you just needed some way to internally edit code in a bigger

[4] http://code.google.com/p/jsyntaxpane/

[5] jFlex is a lexical analyzer generator for Java, written in Java. — http://jflex.de/

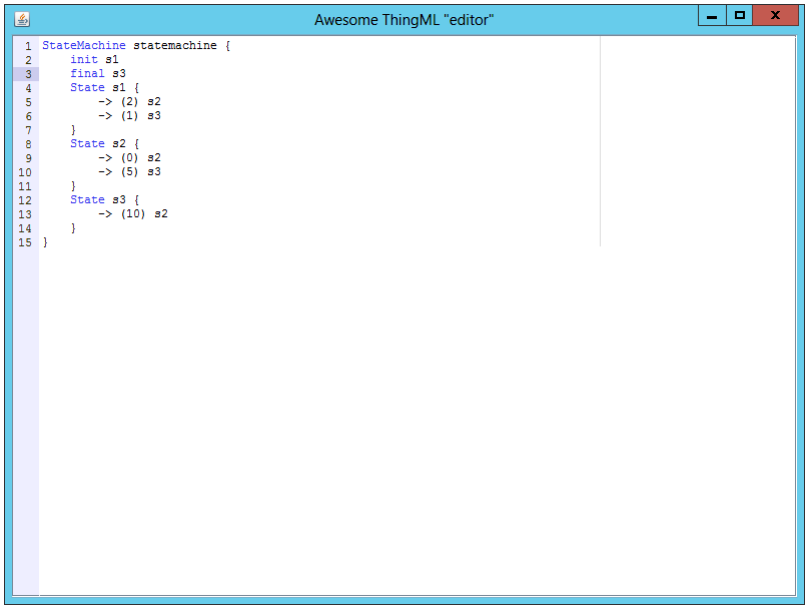


Figure 4.4: The simplicity of JSyntaxTextArea.

project. Like an embed editor. At least it is supporting a lot of the smaller features such as templates, abbreviations, and line numbers.

4.3.3 jEdit

jEdit[6] is the mature programmer's editor as the web page state, and it would be categorised more as an IDE similar to Eclipse then a standalone editor. The advantage of being an IDE comes with having a wast number of features already implemented. And one of the bigger advantages with jEdit is the possibilities for plug-ins. Most of jEdits features comes from plug-ins. And since jEdit is written in the commonly known language Java, it is easy to implement your own plug-ins. If you want to add your own language (also called modes), you only need to create an .xml-file and sort out your operators and keywords similar to how you build a jFlex or ANTLR-file. After which you add your language to a list of all the languages already supported. This is used to tell the editor when to turn on syntax highlighting for a specific language. Other than syntax highlighting it is possible to implement a simple code suggestion called a combo list. To implement your own parser and other more elaborate features you have to write a plug-in.

Looking at figure 4.5 we can see that it is an impressive tool supporting a lot of the minor features necessary for a good tool. With the implementation of other plug-ins it could be a good all around editor for an aspiring developer.

jEdit is a free software so you can use the source code and implement

[6] http://www.jedit.org/

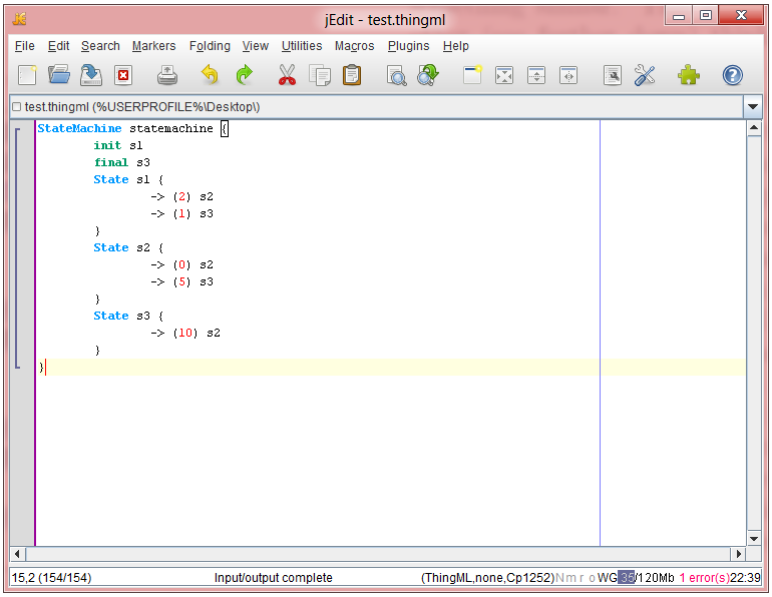


Figure 4.5: The jEdit IDE with ThingML mode.

your own plug-ins, before you compile your own version of jEdit. The way it is buildy up makes this a tedious task, and doesn't really give you the freedom a framework should. In one way, to sum up, jEdit only supports syntax highlighting and code-suggestion out of the box. Other features have to be implemented with a plug-in.

4.3.4 RSyntaxTextArea

RSyntaxTextArea[7], similar to JSyntaxPane, is a Java Swing text component. Originally it was a part of RText, but it has been phased out into its own .jar[8] to be used as a reusable Java Swing application. The benefit with RSyntaxTextArea compared with JSyntaxPane is that it is a part of a bigger project and that it's still being developed. Implementation of your own mode is done by a parser of your own choice. And as with jEdit you need to add your mode to a list of all the supported languages. This is to get it compiled with the archive. Using only the RSyntaxTextArea framework you will just get syntax highlighting and source tree. To add code-suggestion you need to add a different library called AutoCompletion [9] and add them using the Basic Completion class. If you want code-completion you need to implement your own tokenizer and scanner. To test this out, Jflex and CUP[10] where used to create an abstract syntax tree (AST). Using the abstract syntax tree you can create proper error-reporting and code-completion as seen in figure 4.6.

[7] http://fifesoft.com/rsyntaxtextarea/
[8] Java ARchive file
[9] Also developed by Fifesoft.
[10] LALR Parser Generator in Java - http://www2.cs.tum.edu/projects/cup/

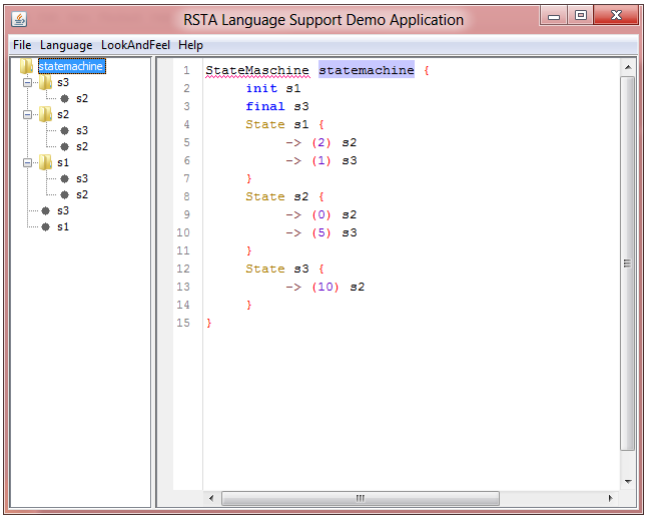


Figure 4.6: RSyntaxTextArea with error-reporting and syntax tree

Similar to Eclipse it also supports source tree, this is mostly because it provides the necessary classes, as long as you write the parser for the text to tree model. Implementing a source tree then is pretty straight forward when you already have an abstract syntax tree. The error pop-up box and squiggle line is all a part of the framework, and there is only a need to extend the AbstractParser-class and use the DefaultParserNotice-class to add the notice and offsets.

In contrast to Eclipse the framework doesn't manage what kind of error is reported and how the user can fix it. In figure 4.6 the user have written 'StateMsachine' instead of 'StateMachine'. To suggest possible solutions you need a method that can scan the error message (a word in this case) and see which keyword it matches, before it gives a suitable feedback. This sounds like an easy task, but if you are using jFlex or similar, the offset given by the scanner is not the same offset that RSyntaxTextArea want, as jFlex provides the offset by line, while RSyntaxTextArea see the whole document as a line.

4.4 Summary

In this chapter we have looked at different tools and frameworks that already exist and could be usable for implementing domain-specific languages. There is a lot of different text editors and integrated development environment (IDE) out there, so we ended up narrowing it down to 11 editors based on feedback from students and professors at Department of Informatics, University of Oslo. Of these 11 editors four where chosen for implementation of the simple version of ThingML called "Simple ThingML". The main focus was on jEdit and RSyntaxTextPane as two new suitable frameworks for the ThingML editor, since both Eclipse and JSyntaxPane had already been tried out as possible editors for the ThingML language.

The reason for looking more into the two already used framework was to see if it would be better to expand what we already had instead of starting a new. From Eclipse we proved what we already knew, that it would be to advanced for novice users. JSyntaxPane could had been a promising framework to continue working on, but since it was no longer in development and it was really lacking in features, we abandon it.

jEdit was looking very promising when we started using it. Mostly because it was widely used and supported, had a lot of plug-ins and was written in Java. Unfortunately it ended up being too complicated and was not as modifiable as it wanted it self to be.

We ended up going for RSyntaxTextArea as we felt it would be a good platform to develop for, also it still under developing and supported by a fairly big and active community. It also has a lot of the features we felt were necessary to get going with a great editor. another great thing with RSyntaxTextArea is that it was easy to work with and modifiable.

Chapter 5

EMFText and XText

In this chapter a discussion of the difference of the two main tools for the development of domain-specific-languages with Eclipse is presented. Before going into detail about Xtext and EMFText, there will be a look at what they both have in common, and which one who fit best with the ThingML project.

5.1 What do they have in common

Xtext and EMFText are both using ANTLR[1] as parser. They both create Eclipse ready plugins with built in parsers. In the end you end up with the same product, it is only two different ways of building a domain-specific language with and for Eclipse. So the question should rather be formulated as what does working with them have in common.

Both tools also gives us more or less the same features;

* Code completion
* Syntax highlighting
* Outline (code source tree)
* Automated parsing and support for quick-fixes and warnings
* Advanced bracket handling

5.2 EMFText

EMFText is built on top of the models generated by Eclipse Modeling Framework (EMF), and uses that model to build a syntax to describe the domain-specific language. Works with the EMF XML-schema Ecore used to define object models. When developing with EMFText the developer needs to be familiar with EMF and needs to make a model of the DSL. Using the graphical Ecore UML class diagram builder is an easy way to build the

[1] ANother Tool for Language Recognition, ANTRL is open source and probably the most known parse generator, at least for LL(*) grammars

model if not generated or given by other systems. Using only the model to generate the language gives you a bracket looking language, see figure 5.1.

Listing 5.1: Simple ThingML in EMFText standard HUTN-syntax

```
StateMachine {
  name : "StateMachine"
  init : S1
  final : S3
  states :
    State {
      name : "S1"
      source :
        Transition {
          target : S2
        }
      source :
        Transition {
          target : S3
        }
    }
  states :
    State {
      name : "S2"
      source :
        Transition {
          target : S2
        }
      source :
        Transition {
          target : S3
        }
    }
  states :
    State {
      name : "S3"
      source :
        Transition {
          target : S2
        }
    }
}
```

You can use the syntax definition tool[2] given by EMFText to define your own syntax by modifying the keywords. EMFText does not come integrated with a code generator or other transformer technologies, but instead lets the user decide which one to use. This is often preferred if there is no need for a code generator or transformer, as you save the time needed to remove the generator or transformer from the generated Eclipse-plug-in.

[2] Referred to as CS-file, Concrete Syntax.

5.3 Xtext

Xtext on the other hands derives everything from a description of a concrete syntax language. In other words, the abstract and concrete syntax is written together. In compilers this is the more traditional way to do it. Therefore there are no need for the EMF model (even though you can generate it using Xtext). You start of by defining your grammar and Xtext generates the rest for you. Adding constraints to the syntax is easily implemented by creating a Java class, since everything is done by dependency injection. Thus Xtext takes care of the generation and interaction with the editor. When it comes to code generation and transformation Xtext uses Xtend and Xpand, so if you want to use another technology you need to de-configure and reconfigure the project. Meaning that if you want code generation or transformation your best bet is to stick with Xtend and Xpand. In summary Xtext uses one file to handle the abstract and concrete syntax. The rest is generated based on that file. In listing 5.2 you can see a "Hello World" example language. The first rule is always used as the start rule. In our case it is 'Model'[3]. It is only function is to say that a 'Model' can contain an arbitrary number of 'Greeting' and store them in 'greetings'. Each 'Greeting' contains a keyword 'Hello' followed by an identifier using the ID which is defined in the super grammar imported at the top of the code[4].

Listing 5.2: "Hello World" example language in Xtext

```
grammar org.example.domainmodel.Domainmodel with
   org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/
   domainmodel/Domainmodel"

Model:
 greetings+=Greeting*;

Greeting:
 'Hello' name=ID '!';
```

5.4 Which one to use

The question then is whether to choose EMFText or Xtext. They both give you the same functionality[5] and are both well integrated with Eclipse, and in our case they are both easily implemented with other editors. Christopher Guntli[7] have made an interesting chart on which task EMFText may be preferred over Xtext and vice versa. See table in figure

[3] There is no name policy

[4] Read more about how to work with Xtext at http://www.eclipse.org/Xtext/documentation.html

[5] In this context I'm talking about the final language, and not features or functionality when developing a DSL.

Task/Situation	EMFText	Xtext
A model is already created	++	+
The model gets generated through another tool	++	o
A database layout is used for the DSL	++	+
The syntax is already given or a predefined scheme	o	++
No model is needed	o	++
Lots of constraints like upper-case letters or maximum entries	o	++

Table 5.1: Comparison between Xtext and EMFText

5.1. In this table Xtext is slightly ahead. This may be because Guntlis' personal choice is to go for Xtext as he likes the way Xtext implements code constraints and formatting by using code injection. If you are already familiar with EMF and the Ecore model, the best choice would probably be EMFText.

In summary the main difference is that EMFText is built up by two attributes, the EMF model which describe the context of the language, and the concrete syntax[6] file implemented in EMFText let you describe the syntax of the language. In Xtext on the other hand both the context and the syntax are described in the same file, and there is no need for the EMF model (that you can optionally generate).

There may be some under laying differences in how the Eclipse-plugin is generated, for example how Xtext handles expressions and priority between operators while keeping the metamodel clean from any ad hoc solution, but this is outside this thesis scoop.

5.5 What we used

In the ThingML project we ended up using EMFText. The main reason for this is that the developers are most familiar with EMFText and as mentioned earlier that is often the main reason to go for one or the other. Another benefit of EMFText is the use of the EMF model which is a graphical metamodel of the context of the language. This can easily used to describe how the context of the language is built up.

[6]The .cs-file

Part II

Research

Chapter 6

Survey of needs

This section discusses the results from the survey about developers needs in an editor. After the survey had been up for 12 hours, nearly 40 different developers had sent in their feedback, which is nearly all of the developers that had received the electronic survey by e-mail. Other communication of the survey was done through word of mouth, whiteboard in lectures, and info screen spread around the Department of Informatics. First it starts of presenting the survey in its full, then present the raw data, followed by a discussion on ambiguity some of the questions arose. At the end of the interpretation of the answers there is a discussion of what this survey teaches is presented.

6.1 The survey

The survey has a total of 5 mandatory questions, and three text boxes to add more information. The first question was to gather information about what level of developing experience the users where on.

Since this survey focused on what editor features developers uses, there was a certain need to find out what tools they already used when developing, and if the features predicted was necessary for an editor. Since it is not only important to find out what features a developer use, but also to see how much use that certain feature has. There was also a table asking for how much the participant used each of the features, ranging from “not used”, ‘uses’, to “uses all the time”. Between what kind of features a developer uses and how much they use it, there was a question regarding what kind of features that they felt was necessary to make a good IDE/text-editor. The where also three text-boxes in this survey to let the participant elaborate to either the features they thought was needed for a good editor, other features a developer use, or just wanted to add some thoughts about this survey or IDE/text-editor in generally.

6.2 The raw data

This section tries to present the raw data from the survey as objective as possible. The complete questionnaire and answers can be seen in appendix 12.1.4 and 12.1.4.

6.2.1 Developing skills

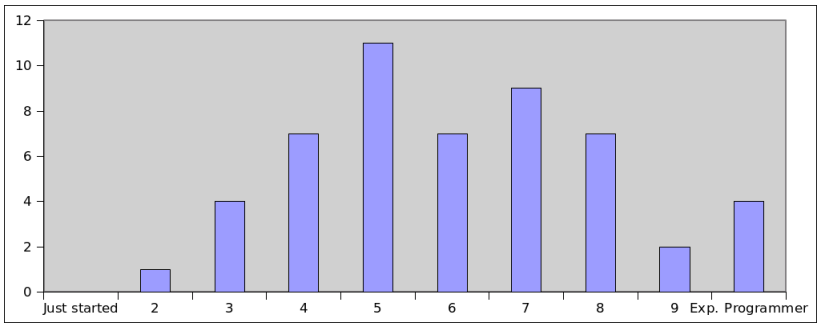


Figure 6.1: Question 1: How would you rate your developing skills?

Question 1 as seen in figure 6.1 was a pretty straight forward question, and we can see that non of the participants had just started, while most of them was in the range from 5 (what would be qualified as competent) and up to 8. While there was 4 persons saying they where experienced programmers.

6.2.2 Text-editor or IDE of choice

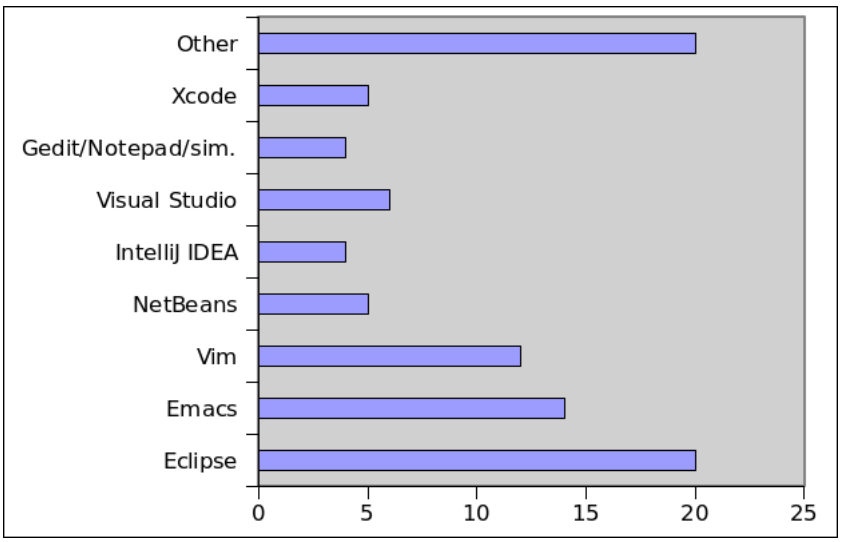


Figure 6.2: Question 2: What is your text-editor or IDE of choice?

To get some grounds on peoples usage of editors or IDE, there had to be a question regarding this. As you can read the single most used editor is Eclipse is the one widely used, followed by "other-group". Both Emacs and Vim both have a strong user base, and combined they range at the top. In the 'other' category Sublime is the editor jutting out. The rest was other more rarely used editors with one vote each.

6.2.3 Supported features

Question 3 was another "getting the ground" type of question. And the answers tells that most of the editor used to support the features asked for. "Source tree" and 'refactoring' where the two with the least votes. Reasons for this are brought up in section 6.3, Ambiguity. Not much to say, other then to note that not everyone is using syntax highlighting.

6.2.4 Necessary features

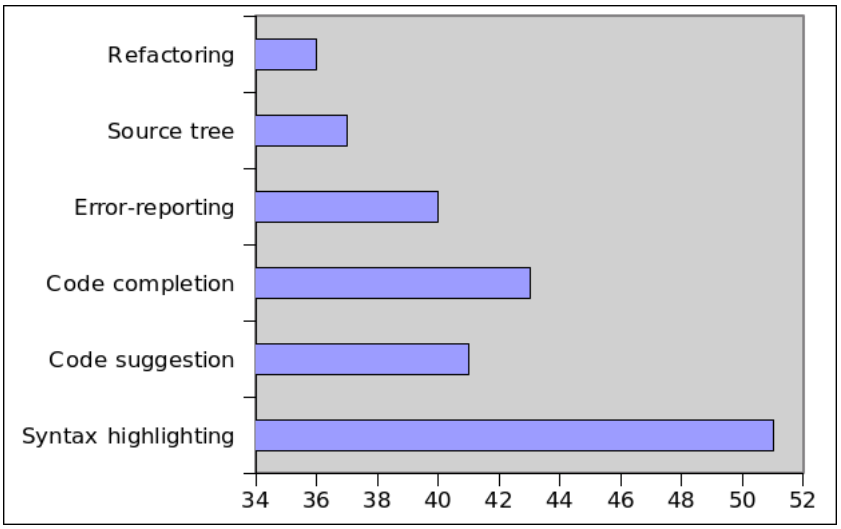


Figure 6.3: Question 4: Which of the following features do you think is necessary for an IDE/text-editor?

Let us continue on with which features are necessary for an editor. Most, but not all voted on syntax highlighting, while the rest of the options, except refactoring, had votes of 71% or more. Thus refactoring was the feature the users felt was least necessary for an IDE/text-editor, just ahead was source tree. There was also one participant who filled out the text-area tied to this question. That person wrote;

> All the rest is very usefully[sic], but syntax highlighting is the only one I can not be without.

6.2.5 Features used

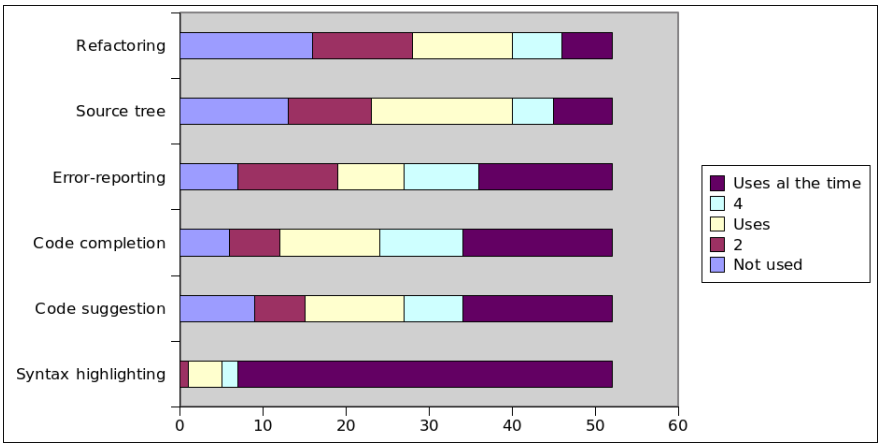


Figure 6.4: Question 5: Which of the following features do you use when you develop?

The last and maybe the most interesting question, is question 5. This is about what features that the participants uses when they develop. Every user said they use syntax highlighting at least rarely, but the majority answered “Uses all the time”.

* The feature “code suggestion” was more divided. Still the majority answered either “uses” or “uses all the time”, while the rest was evenly spread with a peak at “not used”.

* Next is “code completion”, not the same graph as “code suggestion”, but still nearly the same amount of users answered “uses all the time”. The rest were spread out, with ‘uses’, and ’4’ where leading with at least 7% each.

* Then we have “error-reporting” where the usage is more parted again. “Uses all the time” leads with 31%, but ’2’ is close behind with 23%. Again the rest is spread out, with a small drop at “not used”.

* “Source tree” is the first of the features with a huge drop in the amount of use. Still the middle option “uses” gets the majority of the votes. Combining the top two and they reach just above “not used”.

* “Refactoring” has a similar graph as “source tree”, but where ‘uses’ is switched with “not used”. ‘2” and “uses” is also tied, which is a decrease on “uses” from the “source tree” graph.

6.2.6 Elaborated answer and added information

That was the overall answers given to the questionnaire. There where two text-boxes left at the bottom, for the participants to write more elaborate

answers. First box was related to if the participants had other features they used when developing. See attachment 12.1.4 for a complete list of answers. The least topic-specific answers were tabbing and definition jumping[1]. The more interesting answers was that the editor had to be configurable, support customizable key bindings, and the option to turn certain features on and off. Kinda like a *expert-mode* button

The last text-box was there if they had anything more to add.

6.3 Ambiguity

After the survey had been sent out, some of the recipients came to discuss some of the questions and the available answers. It quickly arose that people had not read the transfiguration in the description. This fact in turn led to users answering the questions with different features in mind. Especially 'refactoring' and "source tree" where two features that the users had misunderstood or thought was directed to other features. The obvious reason behind this misunderstanding is that the survey participants did not read the description before they did the survey and did then not get the explanation about the different terms used in the survey. The more elaborate reason is that the two terms 'refactoring' and "source tree" are also used to describe similar and other features.

6.3.1 Refactoring

When we talk about refactoring in generally, developers often think about code refactoring. This is best described by Martin Fowler[2];

> Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

So the case in this survey may have been that the participants had been thinking about code refactoring, which is something that not many tools have native supports, since this often is a difficult task, even for humans. Which would often make it even more difficult to make a feature that would do it automatically. This was certainly not what was meant in this survey. The name refactoring in this case comes from Eclipse, and their use of refactor or refactoring. And it is the Eclipse kind of refactoring which was meant to be asked for in the survey.

6.3.2 Source tree

The "source tree" ambiguity derived mostly from that source in developing are used to describe a lot of different aspects, and that a source tree in some form/kind are something most IDE and some text-editors have. In this

[1] Also requested as "source links", accessing files with ctrl+left click

[2] Read more at http://www.refactoring.com/

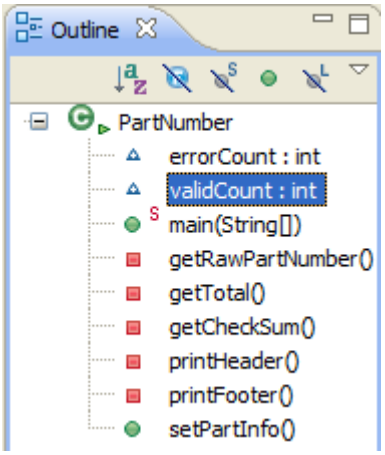


Figure 6.5: Example of a source tree in Eclipse, called 'Outline'

survey a source tree was supposed to represent a tree of the code you where currently writing, but a source tree may in many occasion mean a list of files in a project, also called a "project tree" or "project source tree". In Eclipse a source tree in our context is called 'outline', see figure 6.5. This may have been a better name, and would probably made for less confusion. Other viable terms are abstract syntax tree from compilers, or just plain "code tree".

6.3.3 Minor ambiguity

This ambiguity where not raised by any of the users, but "code completion" and "code suggestion" may have been mixed, or at least cause some confusion, at least for the lesser experienced programmers. The reason for this is that the two features are used interchangeably. It is also combined and used under other names, like Eclipses "Content assist" or IntelliJs Coding Assistance[3]. In this context they are used to describe two different features, as stated in chapter 8.

6.4 Interpretation of the survey

This section includes an interpretation of the survey, and how it is relevant to this thesis. In other words, how it support the thesis. Each questions are listed up, with an interpretation. The first question is brought up in the next section.

* **What is your text-editor or IDE of choice?**

  From the survey we can read that Eclipse is the most used IDE with 38% of the participants votes. This can be a good indication that

[3]Including such features as "Smart Code Completion", "Instant Completion", "Chain Completion", and so on

Eclipse is a good IDE to have as an ideal to work from. Still, on the other side we have Emacs and Vim. Which combined has 50% of the votes. Both Emacs and Vim are two very different editors from Eclipse, and could of course be one direction to aim for. If we look at the 'other' category, which has also has 38% of the votes, Sublime is the most represented. And Sublime as an editor is much more in the Eclipse style, then Emacs or Vim.

* **Does your choice of IDE/text-editor support one or more of the following features?**

  Next question was used to both see what features their editor had, but also to see what kind of features the participants knew their editor supported. It is interesting to see that 8 of the users that said they used Eclipse was missing at least one of the features that Eclipse supports. There were also no consistency on the level of programming experience and features that they ticket of on. Even though in generally the participants with lower programming experienced ticket of fewer features. There were also some inconsistency between "code suggestion" and "code completion", which probably stems from the ambiguity between them. At least what can be read from this question is that most of the editors supports a wide range of the features.

* **Which of the following features do you think is necessary for an IDE/text-editor?**

  Now that we know what kind of features programmers have in their editor, it would be interesting to see what kind of features they are looking for. The answers received lined pretty nicely up with what features the editors/IDE they already used supports and with our predictions. By looking at the next question, we also got some kind of indication on what features we should focus on.

* **Which of the following features do you use when you develop?**

  Last question was about which features they used when they where developing, and how much they used each specific feature. As predicted most of the features were frequently used, and as previously questions shows, "source tree" and 'refactoring' where the two that was least used. This, again, can be either because of misunderstanding of what the survey meant, or that they just didn't use it that frequently. "Code completion" and "Code suggestion" had also nearly the same answers, which is likely since most editors/IDE that have code suggestion, also support code completion. Still there were some discrepancies of editors who had both features, but still only got a vote for one. "Error-reporting" on the other hand has a more divided feedback, where over 30% says that the use it all the time, but over 30% says that they use it rarely (which is indicated by a 2 in the survey) or never. The remaining 30% was spread between '4', and 'uses'. Compared to what kind of editor/IDE participants uses, the distribution

between “uses all the time” and “rarely used” is comprehensible, as Vim and Emacs (with 50% of the participants uses) does not support error-reporting by default. As they don’t have any understanding of languages without modes or scripts. So what can we derive from this question is that we should focus on syntax highlighting, code completion and suggestion, and error-reporting as our main features in the new editor. “Source tree” and ‘refactoring’ based on usage is not that important. This is also supported by which editors the participants have voted on, and which features they say the editor support or that they use.

As mentioned in the raw data section, there was a text-box for the participants to add features that they did use, but which were not mentioned in the survey. A common denominator of the extra features mentioned, are that they are complex and split between different tasks. Also none of the participants mention the same feature. Some of the more difficult and complex ones, which where not tied to language specific features, were; debugging, version control, plug-ins and plug-ins manager, code optimization[4], and macro-recording. Features that where suggested which are possible with the RSTA framework; definition jumping or source linking[5], difference view, keyboard hot-keys, code generation, dynamic templates, and split window.

There was also mentioning of features that are already functional in RSTA, which were not brought up in the survey. Such as; automated indentation and tabbing. Both of these could have been added to the survey, but were forgotten.

6.5 Threat validation of the survey

The main validation problem with this survey is the response size. With only little under 50 responses, the margin of error can be quite high[10]. Still, if we take a look at Eclipse usages data[6] can see similar feedback. Similar data can also be retrieved from NetBeans[7] we can see that for example “content assist”[8] is the fourth used command, behind ‘paste’, ‘save’, and ‘delete’. So if there would be another research on this topic, using the same methods and question, it is believed that the outcome would be nearly the same.

Let us take a look at question 1 from the survey, which is about the level of experience of the participant. As you can see in figure 6.1 none of the replies said they had just started, but some where fairly new to the game, while 80% feel they are competent to experienced programmers. This gives

[4] Similar to code refactoring

[5] Accessing other files via ctrl-click

[6] See http://www.eclipse.org/org/usagedata/

[7] See http://statistics.netbeans.org/analytics/

[8] Eclipse version of a combination of “code completion’ and “code suggestion”. With the id “org.eclipse.ui.edit.text.contentAssist.proposals” in the usage data

us a clear overview of the user group, and with their answers a fairly good representation of what tools and features developers uses.

6.6 Summary

When using the data from this survey it is important to remember the low response rate, which makes the margin of error quite high as mention in section 6.5. There is also the problem with ambiguity and there was even a comment on that there where no description of the terms used in the survey which may have shifted at least the use of source tree.

To continue with the importance of lightweight editors, we can see that nearly all of the participants use a lightweight editor, in some way. Most of the participants that only uses one lightweight editor are either using Emacs or Vim. Of the participant that also uses IDE's, a lot of them uses Sublime. One common denominator between the lightweight editors are that they are highly customizable. Other features the lightweight editors support are tabbing and code templates.

The goal of the survey was to find out what features this thesis should focuse on. Based on the answers there are certain features that are used more then others. From figure 6.4 we can see that the top most used features are syntax highlighting (in the clear lead), followed by an even use of code completion, suggestion and error-reporting. Source tree is a bit far behind, but that may mostly be because of the ambiguity that was discussed in section 6.3. It is also interesting to see that (if we exclude syntax highlighting) error-reporting is the feature with the least "not used". Of course when choosing which features to implement there is also a cost-benefit ratio. So based on this survey there is a focus on implementing the following features; syntax highlighting, code completion, code suggestion, and source tree.

One feature not brought up that is mentioned several times in the "Anything more to add?" column is the need for good and customizable keyboard shortcuts. some users also want key bindings used in Emacs and Vim. This is something that should be thought of when developing as it will make it easier for developers to change editors when working with different languages/editors.

Chapter 7

Observation of Eclipse

To get more hands on data about how users may utilize Eclipse there where held a series of Eclipse observation workshops. Each session lasted 60 minutes, after which every participant had to answer four questions about the participants use of Eclipse.

1. Which level of programming experience will you say you are on?
2. How long have you used Eclipse?
3. Have you got any teaching in the use of Eclipse?
4. What do you think of Eclipse as an IDE?

The plan was first to sit and observe while the participants worked with a project in Eclipse. After first session it was made clear that this would not be sufficient, at least for the novice Eclipse users. So the session got split into two parts. The first part was a normal observation, while in the second part, the participants would get a set of task to complete. Each task focused on different aspects of Eclipse and its features.

The task that the participant conducted utilized the different features that was presented in the survey in chapter 6. Following is the list of each feature, and the task related;

Syntax Highlighting This is more of a question of the understanding of what syntax highlighting is.

Content assist First some informal question about what code completion/suggestion is, and if the participant has used it before. Then there are some tests where the participant needs to write a method using code completion.

Error-reporting Based on how the participant uses error-reporting in the first part, there is also some light discussion of usage of it. In addition there is some test to see if certain problems can be solved with the error-reporting in Eclipse.

Source tree Next task focused on the source tree. How to use it, and how it may speed up the developing process.

Refactoring Last task is about Eclipse refactoring, and the user where given two task. One where the participant shows me how he/she tried to change several variable names, first in one document, then in several. In the second task the participants where taught how to use refactoring in Eclipse.

7.1 The observations

7.1.1 Observation one

First observation was done with a bachelor student working on a mandatory task in Java programming, which would increase the reliability of the observations. It was decided to change the observation to use that same mandatory task for each observation. Since everyone then had do the same task, it would not be a maid of task to either unknowingly shift to the observers advantage. The participant had also just started using Eclipse this semester, after a tip from a friend. The observation was arranged so that I was sitting behind the participant, looking at the interaction with Eclipse while the participant tried solving the mandatory task. There was no guidance from me on how the participant could utilize Eclipse or features that could possible be used. The whole session took a little bit over an hour.

7.1.2 What was observed

As the participant started to work on the task it was quickly made clear for the observer that the user nearly never utilized any of the features in Eclipse. The only widely used function was “content assist”. And even this usage wasn’t that consistent. What often happened was that the participant used the mouse to copy/paste an earlier line, instead of using content assist to help him write or finish variable names. One feature the participant did use a lot was code templates. More precisely one template, this was the ‘syso’. When typing ‘syso’ and hitting “ctrl+space”, Eclipse fills out “*System.out.println()*” for you. The participant had gotten this trick from a fellow student.

7.1.3 Observation two

The second observation was a bit different form the first in that the first observation alone didn’t give much feedback on how the user could work with Eclipse. So based on that case the second observation was split in two. The second participant was also a novice to programming and the use of Eclipse.

7.1.4 What was observed

In the first part the participant was solving the mandatory task. As predicted, and which happened in the first observation, there wasn’t that

much to gather from the observation in the use of Eclipse. What the participant did use was the error-reporting and the outline. Outline was used a lot to move around in the code. Error-reporting was only used to see what the error was (mostly because there would be a red squiggly line underneath). There was no usage of the quick-fixes supplied underneath the error message. The participant did try one time to use the Eclipse debugger, in order to solve a compiler error in code, but quickly gave up. The other features where not used at all. Also, instead of content assist, copy/paste was used instead.

In the second task, based on what had been observed, the participant was asked to perform certain tasks with some of the features that the participant had not used in the first part.. The features that were picked out was content assist, refactoring, and the quick-fixes for error-reporting. The task was presented in a way where I as the observer first explained the feature and what it could be used for, then the participants was asked to perform to utilize it.

7.1.5 Observation three

The third observation was with a more experienced programmer and Eclipse user. To keep all of the observations as similar as possible, the participant, as mentioned, was asked to perform the same mandatory task as the two previous participants where working on. Since this was a rather easy task for the third participant the whole session took only about 30 minutes to perform. Also as the previous observations I as the observer was sitting behind the participant and observed how the participant was using Eclipse.

7.1.6 What was observed

The third participant had some trouble understanding the mandatory task given, but after some clarification, everything went smooth with the developing. What was observed was that the participant did use some content assist to write code quicker, but it was not every time. So it looked rather randomly. The participant also used error-reporting and quick-fixes rather efficient, which accelerated the coding of the task. There was no observation of use of template to save time writing methods or classes, and no use of outline. The participant also chose to only use one file for the whole mandatory task, even thought it included different Java classes.

7.1.7 Observation four

The fourth participant was another competent Eclipse user, and since he was doing the course with the mandatory task (which he already had done), the Java part of the observation went rather smoothly, which gave a better flow in the use of Eclipse since most of the time was pure coding and not fiddling with the understanding of the mandatory task. Similar to

participant three, this session took little over 30 minutes to complete, and was done with me as the observer sitting behind the participant.

7.1.8 What was observed

This participant had no problem understanding the mandatory task, and got started quickly with the task. Similar to participant three, there was no troubles with the Java. The user was frequently using templates to create methods, but they all ended up being private. So the participant had to change them all the time to public. This was solved by a lot of copy and pasting. Other than code templates, error and warning reporting (displayed as a red or yellow squiggly line) were used a lot, but as a lot of the other participant most of the errors where read while hovering the mouse over the line column (which also show the error). There was also no use of quick-fixes. Another usage which was not present in the other participants was the use of indent all, which work with first selecting the area to indent, and then pressing the corresponding hot-key. This sometimes led to lines not being indented, even though the participant thought so. Which sometimes let to minor frustration. Last feature that was observed was content assist. It was mostly used to speed up writing of method names, and rarely used to finish variable names.

7.2 Summary of the answers

After each observation session, the participants were ask to answer four questions. The answers in their written form can be found in appendix 12.1. The following is a discussion of the answers given. The level of experienced range from beginner to something between medium to expert. One participant says the level is medium compared to the classes the participant have done, which doesn't really states what level of experience the student is at, but based on the observation, I would qualify the student as a beginner, similar to the second participant. While participant three and four were both competent programmers.

Of the four, only one of the participant had used Eclipse over several years, while the other ranging from just a couple of weeks to months.

Only one of the participants mentioned that he has received some form of training in Eclipse, but it was mostly focused on modules and simulators in Eclipse. One mentioned using Google when needed.

There is an unanimously agreement that Eclipse is a good IDE. Some feel it makes the developing easier. Still they mention a lot of challenges using Eclipse. Two of the participants mentioned trouble of understanding the programming environment and project managing. For example they end up creating new classes in the wrong project and similar. There is also the challenge of navigating the Eclipse interface for beginners, and one felt it is made for *experienced users and is not suited for beginners*[1]. One

[1]See answer 12.1.2 and 12.1.4

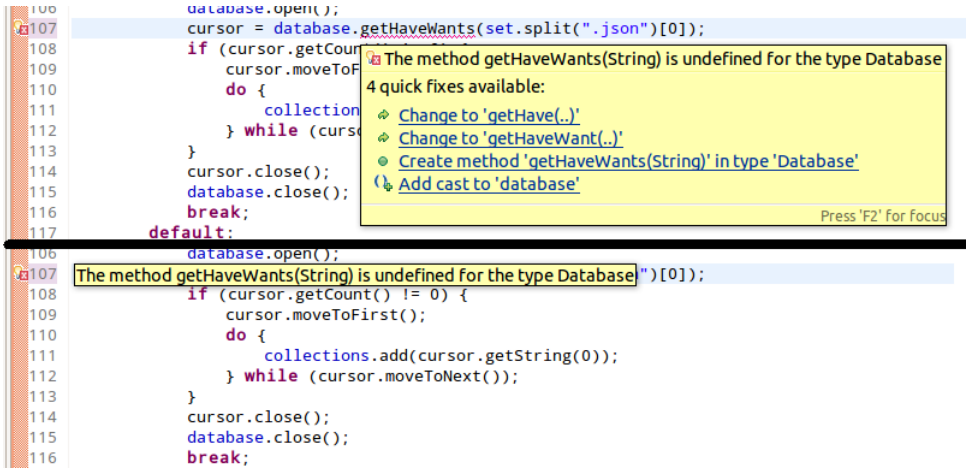


Figure 7.1: The two kinds of error-reporting in Eclipse

feature request for a new version of Eclipse is the possibility to toggle the advanced features on and off.

One of the participant who is a beginner in programming also notes that it is important to not get too attached or dependent upon using Eclipse, in case you start to forget basic programming that Eclipse automates for you[2].

7.3 Discussion

It was interesting to see how the different users interact with Eclipse, even though the experience level was not as widespread as was wished for in this type of observation. When looking back at the participant there where no expert users of Eclipse. Of course the task asked to be done may have conjured the low usage of necessary features, as the coding part of the observation was a fairly easy task for the more advanced Java developer.

One thing that was interesting is that there was no usage of quick-fixes from any of the participant. This is probably something that an advanced user would rely heavily on to fix the minor errors, typically in spelling or creating methods. This may have been because most of the participants didn't even check the squiggly line underneath the text, but instead checked the line column where the error symbols are shown. This message only show the error, and not the suggestions for possible fixes. The two different errors messages can be seen in figure 7.1.

So to sum it up the novice user did not use or utilize that many features that Eclipse supports. There may be a lot of different reasons for this, but it probably mostly because of the lack of understanding or knowledge of the features in Eclipse. This is also something that is widely discussed in the paper "Experiences with Eclipse IDE in programming courses"[2], where the writer examine the usage of Eclipse IDE in programming course for first

[2]See answer 12.1.1

and second semester students. They found out that to have the students utilize Eclipse in best manner they had to teach the students slowly how to use Eclipse. The conclusion was that the best move was to not use Eclipse at all in the first weeks, and then gradually teach the usage of Eclipse in the group classes while they were teaching programming.

Chapter 8

Needs and requirements — The important features

In this chapter the six features that were implemented with the new framework is presented. Each of the features are based on the combined findings from both the survey and the observation.

8.1 The six important features

Each feature is represented, starting with the easiest and ending with the most complex one.

8.1.1 Syntax Highlighting

The most basic feature any editor needs is syntax highlighting. Luckily syntax highlighting is a relatively easy task to implement, mostly because all what is needed is something that can parse the text and color it ongoing. Something as simple as regular expression could be used for parsing. Figure 8.1 shows an example of Simple ThingML where both the keyword 'StateMachine' and 'State' have been colored blue. There could also been a different color for variables such as 's1'.

Listing 8.1: Using syntax highlighting the keyword 'StateMachine' and 'State' is colored blue.

```
StateMachine statemachine {
  State s1 {}
}
```

8.1.2 Code-suggestion

Another simple feature needed for the editor is code-suggestion. It is important to not misunderstand code-suggestion with code-completion in this context. In this context code-suggestion is meant to be about finishing and suggesting the chosen languages keywords, not variable names and

methods, which is done when using code-completion. Code-suggestion is implemented with similar methods as syntax highlighting. All needed is a list of all the keywords in the language and to look them up as the developer type. Often there is a threshold of a couple of letters to avoid flooding the user with suggestions. A more complex code-suggestion also uses previously typed word either in the session or in the file to help populate the suggestion list. An example of code-suggestion can be made with figure 8.1 at page 51. When the developer start typing 'Sta' both the keyword 'State' and 'StateMachine' can be suggested to the user. Usually the list is alphabetically sorted, so 'State' would always be first. Another example is if we remove the threshold of three letter for suggestion to pop up, then we could have added the variable 'statemachine' to the list when the users starts to type the 's' after writing 'State' on line 2.

The simplicity of syntax highlighting and code-suggestion makes them both supported in nearly all of the lightweight frameworks out there.

8.1.3 Source tree and outline

Another rather simple feature to add is the outline or the source tree. An outline is a summary of the code represented in a hierarchical list, often represented using the 'class' as root followed by variables and variables type and functions.

Implementing a source tree can be done in several ways, depending on your framework. The simplest solution is to use the information the parser for the syntax highlighting provides. If code-completion or error-reporting also is implemented, using the data provided with those would probably be a better and easier choice. What is important is to have an overview of every variable name and type, functions, and classes. Figure 8.1 shows an example of an outline in the Eclipse plug-in for ThingML.

8.1.4 Code-completion

Code-completion is an extension of code-suggestion in the way that it tries to guess what variable or function you wish to type. This is based on what you have already written. Listing 8.2 shows two state machines with one and two states each. When using code-completion the suggestion that will be presented when writing the 'init' in the state machine 'twoStates', will give you both 's1' and 's2', while writing 'init' in the state machine 'oneState' will only yield 's1'. This is because the code-completion 'understands' the code in the way that it sees that from the scope 'oneState' it can only see one state, namely the 's1'.

The 'real' ThingML as opposed to Simple ThingML is using something called a configuration-file where it looks after which port in the Thing you are deploying your code to, to communicate. This is not that easy to parse using just normal parser, and in this context is is solved by having a meta-model of the project in the computers memory, this is to support an easy look up of earlier written code[1].

[1]Think variables, methods/functions and objects.

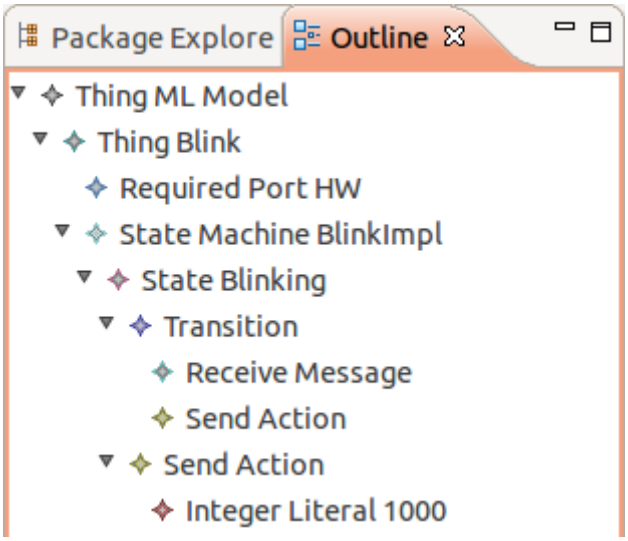


Figure 8.1: Eclipse outline of the ThingML language

Listing 8.2: Code complex enough to be use for code-suggestion and refactoring

```
StateMachine oneState {
  init s1
  final s1
  State s1 {
    -> (0) s1
  }
}

StateMachine twoStates {
  init s1
  final s2
  State s1 {
    -> (0) s2
  }
  State s2
  -> (1) s1
}
}
```

8.1.5 Error-reporting

The point behind Error-reporting is self-explanatory, as it is there to inform the programmer when he/she has either a syntax or semantic error. This can be solved by different methods, but the most common is do some kind of parsing behind the scene. Similar to the code-suggestion and source tree. The biggest difference is that the parser needs to understand what is wrong. A typical error-reporting can be seen in figure 8.2. Here the user have tried to reference a state called 'Blinking2' that is not declared. This in

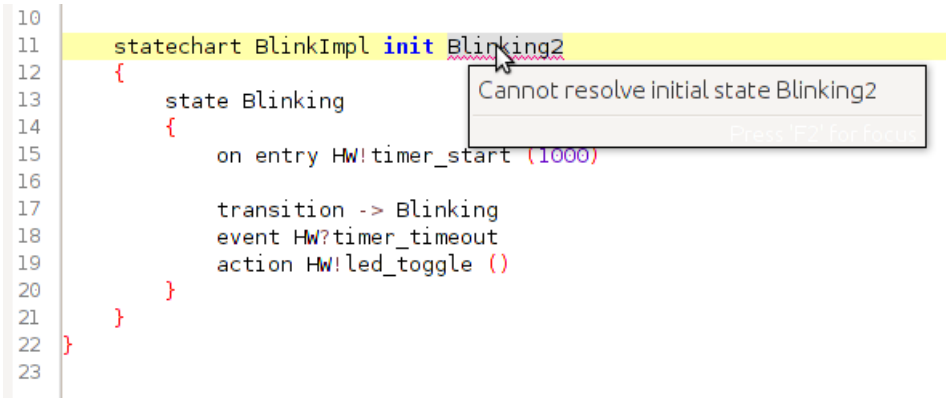


Figure 8.2: Error-reporting in the new ThingML Editor

turn returns “Cannot return initial state Blinking2” from the compiler.

8.1.6 Refactoring

Refactoring is probably the most complicated feature to implement as the editor need to have an understanding of what you have written, similar to the error-reporting. The main difference is that it has to be able to refactor the code by itself, and cannot rely on the user to fix the potential errors that may happen. For example, when renaming a global variable called ‘key’ to ‘globalKey’, the parser needs to see the difference of the ones that you have locally declared and the one that is the true global variable. This is turning the framework more complex as you need to have a detailed description of your language and the code that is written.

An example of name refactoring can be seen in listing 8.2. There there are two different Statemachines with three different states. State machine one (oneState) has one state called ‘s1’. State machine number two (twoStates) has two different states called ‘s1’ and ‘s2’. Without proper refactoring a normal “find and replace” method would interpret ‘s1’ in ‘oneState’ the same state as ‘s1’ in ‘twoStates’, which is not true. They are two different states, and should be treated like that in renaming using refactoring, code-suggestion, source-tree and error-reporting.

8.1.7 Feature summary

To sum it up, these are the features wanted in the ThingML editor framework and why (they are sorted on the difficulty for implementation):

Syntax highlighting Syntax highlighting is implemented in nearly all editor frameworks. As a feature it helps the user to get an overview of the code, and eases the writing in a programming language by making the structure more visually distinct.

Code-suggestion A lot of frameworks support the feature of having a list of reserved keywords tied to a specific language. It is there to help the

developer to see which keywords are reserved in a language, and to help speed up the development time a bit.

Source tree Similar to highlighting it is there to help navigate the code. This is done with a tree representation of the code.

Code-completion An extension of code-suggestion where the framework has a stronger understanding of what the developer is writing. This is used to speed up code writing.

Error-reporting Helps the user to correct minor or major errors in the code on the fly. Often helpful if the variable name is wrong or when type casting is wrong. Example; casting from a *float* to an *integer* will often be reported as a loss of precision error. It can also be very helpful to have if the compile lack detailed error-reporting.

Refactoring Refactoring is something only the most complex framework support (example; Eclipse, NetBeans, Visual Studio (with Visual Assist) and more). Since it forces the editor to have control of all the files in your project, this is because using refactoring on either a file, class, or a method name, can effect several files. Which then the editor needs to automatically change. This feature is there to help the programmer and provide an opportunity to change a name after it has been used in a project without manually having to go through all the files in the project.

8.2 Discussion

The real question when adding all this features to the editor, is if it still would be a lightweight editor, or if we are crossing the boarder into a more slower and complex IDE? This is something that we want to avoid, as the goal is to make a lightweight editor.

It is also important to remember when implementing all of these features that the user interface should not become cluttered for the developer to work with. This is mention in a previous chapter, but it is also important to avoid to have too many buttons that the developer don't really know what to do. The phrase

> Less is more[2]

is something to have in mind when implementing the six features. The less user interface space and actions required by the user, the better the user experience he or she will have. Eclipse on the other hand has gone the other way around. They have a lot of features implemented, and a very complicated user interface with different 'perspectives'. Each 'perspective' shows a different side of Eclipse, and support different needs. The two most

[2] From the poem "Andrea del Sarto" by Robert Browning

used are the Java perspective[3], and the Debug perspective[4][12]. All these layers or 'perspective' makes the IDE very complex and complicated to use, especially for beginners.

There are a lot of features that the future editor could have had, but research shows, if we keep using Eclipse as the example, that there are only a few developers that use each of the features. Meaning that there are only a hand full of features that every Eclipse-user uses. This is also discussed in chapter 6. Another source for Eclipse usage data is the article "How developers use Eclipse IDE"[12]. What's interesting to read is that there are only two commands on the top 10 commands list that are development specific, those are "Content assist" and "Step (debug)". The rest are normal text editing commands such as 'delete', 'save', 'paste', and 'copy'. There are no clear conclusion for this paper, but they do note that there haven't been such research done before on the Eclipse IDE, and that they hope it can be valuable for tool builders targeting Eclipse.

[3]Show the editor, outline, project management and more.

[4]A rather advance Java debugger with a breakpoint view, a dynamic call stack, editor, and a console view.

Part III

Implementation and conclusion

Chapter 9

Building and implementation

In this chapter the process of building a domain-specific language from the ground and the implementation of it will be presented. In this scenario we will be using Eclipse Modeling Framework with the use of EMFText for the concrete syntax. For the editor implementation the RSyntaxTextArea framework will be used.

9.1 The process and steps behind making a textual DSL using EMF and EMFText

There are several tools to use when making a new domain-specific language (DSL). One of the more popular ones is to use EMFText with Eclipse Modeling Framework (Eclipse MF). As mentioned earlier, EMFText builds on the Ecore model produced by Eclipse MF or other tools that can generate an Ecore-model. An Ecore-model is an XML-schema describing object models. The definition used by the EMF project[1] is this;

> The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and run time support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

So the way to make your own DSL is to first download Eclipse MF and install the EMFText plug-in[2], after that you can start modeling your language using the Ecore-model. This can be done in two ways, either by using the Ecore Model editor or the more graphical way with the Ecore Diagram editor.

[1] http://www.eclipse.org/modeling/emf/

[2] See http://www.eclipse.org/ and http://www.emftext.org/

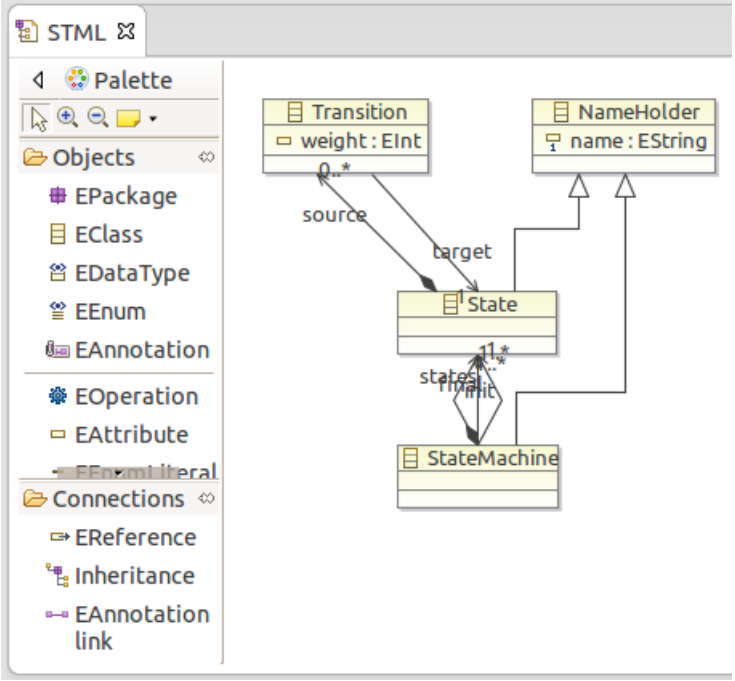


Figure 9.1: The Simple ThingML in the Ecore Diagram editor

9.1.1 Ecore Model and Diagram editor

Using the Ecore diagram (as seen in figure 9.1) is quite different from the model editor as it is an drag-and-drop tool which automatically generates an .ecore-file. This is also the easiest way to work as you will get a better overview of the context of your language since it is laid out in a graphically view with boxes and arrows. To start creating your language you drag out classes which in turn will represent keywords in your syntax. To get the classes to communicate you create reference arrows between them. If you want a keyword to reference or contain another keyword you need to check "Is containment"[sic].

In the Model editor seen in figure 9.2 the building of the language is done by adding children or siblings of the different 'Etypes'[3]. Each children or sibling may then be further modified or enhanced in the 'properties' window.

9.1.2 Generating code

Next part of your domain-specific language is to either create or update the Generator Model[4], which is used to control code generation for the model. The user load the Ecore with the GenModel by right-clicking and selecting 'reload'. Then choose "Ecore model", and browse for your Ecore Model

[3]Child: EAnnotation, EOperation, EAttribute, EReference. Sibling: EAnnotation, EClass, EData Type, EEnum, EPackage

[4]Shown as .genmodel

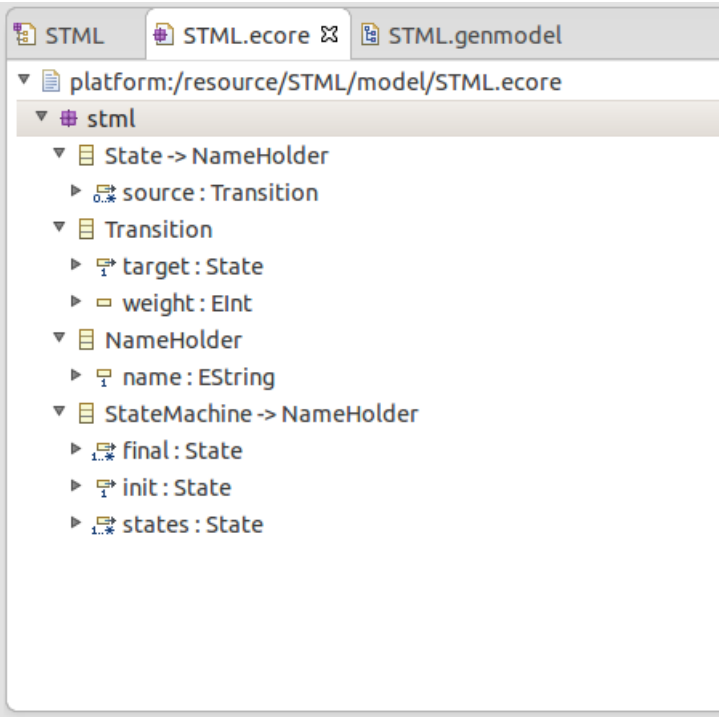


Figure 9.2: The Simple ThingML in the Ecore Model editor

before the user press 'Load'. This will load your new Ecore model into the generator model, which in turn will let the user generate model, edit, editor, and test code for the domain-specific language.

The generated code is used to build the editor and a parser for your DSL. "Model code" is used to generate code implementation of your language, and the interface. 'Edit-' and "editor code" is for the Eclipse plug-in. The 'editor' is the UI for the Eclipse editor and wizards, while the 'edit' code includes adapters that provide a structured view and perform command-based editing of the model objects[5]. If you only want to implement your own editor, there is still need for the "edit code" as it generates a very powerful code-completion interface.

9.1.3 Concrete syntax

Without the use of EMFText and only using Eclipse Modeling Framework it will give you a graphical editor similar to the Model Editor in figure 9.2 page, where you modify or program in the language by right clicking and adding children or siblings.

Using the EMFText plug-in you are able to generate a simple syntax for your language based on keywords. Listing 9.1 shows how the automatically generated syntax looks like, using curly brackets and colons. Using

[5]For more information about generation an editor for the model see http://help.eclipse.org/juno/topic/org.eclipse.emf.doc/tutorials/clibmod/clibmod.html?cp=20_1_0#step3

EMFTexts concrete syntax you can define the syntax any way you want by describing the 'RULES' using the keywords from your Ecore model. It will nearly look like how the developer can utilize the language.

Listing 9.1: Concrete syntax rules for Simple ThingML

```
RULES {
 State ::= "State" "{" ("name" ":" name
    ['"','"'] | "source" ":" source)* "}";
 Transition ::= "Transition" "{" ("target" ":"
    target[] | "weight" ":" weight[])* "}";
 NameHolder ::= "NameHolder" "{" ("name" ":"
    name['"','"'])* "}";
 StateMachine ::= "StateMachine" "{" ("name"
    ":" name['"','"'] | "final" ":" final[] | "
    init" ":" init[] | "states" ":" states)*
    "}";
}
```

An example of a modified concrete syntax can be seen in listing 9.2. Just by moving the name[] variable outside the parentheses we make it a more stricter language, which will help the content assist in Eclipse to be a lot more precise. This is because the content assist reads the group of sub-rules similar to regular expression, interpreting the asterisk as zero or many, meaning that the content assist will think you can write either of the variables in the group as many time you want. This is not something the model support. So to make it a stricter syntax we modify the group, and split it into separated groups.

Listing 9.2: Modified concrete syntax rules for Simple ThingML

```
RULES {
 State ::= "State" name[] "{" (source)* "}";
 Transition ::= "->" "(" weight[] ")" target[];
 StateMachine ::= "StateMachine" name[] "{" ("
    init" init[]) ("final" final[]) (states)*
    "}";
}
```

We also made it easier to code in the language by removing the many 'colons', and the need for using "quotation mark" around names.

With these modification we end up with a language which will look like the one in listing 9.3 instead of what you first had to write in listing 5.1 page 30.

Listing 9.3: The modified Simple ThingML language

```
StateMachine statemachine {
 init s1
```

```
  final s3
  State s1 {
-> (2) s2
-> (1) s3
  }
  State s2 {
-> (0) s2
-> (5) s3
  }
  State s3 {
-> (10) s2
  }
```

When you have defined your own syntax for the domain-specific language you can generate the Eclipse plug-in with the syntax defined in the CS-file. Then you can either run it or use the generated code for implementation in a separated UI. Which brings us over to the next subject, how to implement a DSL made in Eclipse MF and EMFText with a custom GUI or framework.

9.2 Implement of a domain-Specific language with RSyntaxtTextArea

Implementing a custom domain-specific language (DSL) with an IDE or text editor may be a time consuming task. Using the model generated by Eclipse Modeling Framework (Eclipse MF) will make that easier. As last chapter mentioned, the user could use the plug-in that Eclipse MF generates, but this section will show how to implement a DSL with a custom workbench. In this case it will be showed by using RSyntaxTextArea (RSTA) by FifeSoft[6]. The aim was to implement the model generated by EMF with RSTA with changing as little as possible of the RSTA base code. Because of this we are depending on using JFlex to define the syntax highlighting. This may be seen by a drawback for RSTA as you now have two places to manage the keywords used in the language. Since a change in the Ecore-digram and CS-file will potentially affect the JFlex-file. Luckily this is a minor drawback compared to what RSTA gives us for free as a framework.

9.2.1 Implementation of the Eclipse Ecore model with RSyntaxTextArea

To implement your DSL with RSTA you need the model code and model editor generated from Eclipse MF, and RSyntaxTextArea and RSTALanguageSupport (which in turn need the AutoComplete library). The RSTA files can all be downloaded from FifeSoft. From the AutoComplete-project only the JAR[7] is needed. Other then that JFlex is needed to generate the

[6] http://www.fifesoft.com/
[7] Java ARchive

Java-parser for the syntax highlighting. The whole process of implementing the Eclipse model into RSTA is a comprehensive task, and it will be split into different tasks, to make it easier to follow and understand.

1. First separate all the files into different folders, so that you have it all nicely sorted. This will make it easier to update each of the individually part if you need to expand your language, or there is an new update for RSTA. As mentioned earlier you only need the JAR of AutoComplete, so that folder is not really necessary after you've used ANT to create the JAR.

2. Next step is to get syntax highlighting to work. In RSTA this is done by a JFlex file with your keywords linked to the different keywords supported in RSTA. This can of course be modified so that you can add more colors. Listing 9.4 show how the reserved words in ThingML are added to the jFlex-file.

Listing 9.4: Part of the ThingML jFlex file

```
<YYINITIAL> {
  /* Keywords */
  "thing"        |
  "includes"     |
  "datatype"     |
  "enumeration"  |
  "sends"        |
  "receives"     |
  "port"         |
  "provided"     |
  "required"     |
  "message"      |
  "property"     { addToken(Token.
     RESERVED_WORD); }
}
```

Adding languages in editors often called a mode, and can in RSTA languages can be found in the mode folder inside RSyntaxTextArea. Next you need to parse the JFlex-file to generate a Java-file and remove two methods that have been added twice.

> This implementation was created using JFlex 1.4.1; however, the generated file was modified for performance. Memory allocation needs to be almost completely removed to be competitive with the handwritten lexers (subclasses of AbstractTokenMaker, so this class has been modified so that Strings are never allocated (via yytext()), and the scanner never has to worry about refilling its buffer (needlessly copying chars[sic] around). We can achieve this because RText always scans exactly 1 line of tokens

at a time, and hands the scanner this line as an array of characters (a Segment really). Since tokens contain pointers to char arrays instead of Strings holding their contents, there is no need for allocating new memory for Strings.[8]

So to summarize you need to remove the second definitions of both the zzRefill() and yyreset().

3. Next step before highlighting is done is to add your chosen syntax to the file *SyntaxConstant.java* in package *org.fife.ui.rsyntaxtextarea*. When this is done, what is left of RSyntaxTextArea is to make a JAR for the RSTALanguageSupport. As with the AutoComplete this is done with ANT.

4. The rest of the implementation of the Eclipse model is done inside RSTALanguageSupport. API specification used for the code completion for your specific language can be added to the data folder following the syntax of CompletionXml.dtd. Examples can be seen by looking into one of the many languages that are already there. Also before continue be sure to add all necessary libraries to the ANT-file.

5. Depending on what kind of features you want, there are a few steps to take just to implement your own DSL. Start by adding the DSL to the language drop-down menu in DemoRootPane, so that it is easily loadable when the editor is running. This means that you need to have added a text-file with a sample of your DSL code. Figure 9.3 shows the example code used for C. It is only to show the language in a simple text area. On the left you can see the blank area used for 'Outline' or "Source tree", for implementation see step 10.

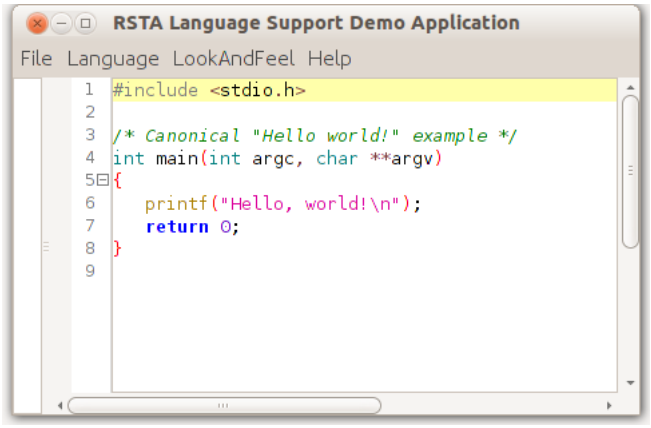


Figure 9.3: A simple RSTALanguage editor with a C "Hello World" example

6. Next step is to add your languages parser, and that means adding your own language factory. This is done in two steps, first add your

[8] This is also mentioned in the newly generated jFlex-file

language to the *LanguageSupportFactory.java*, and then creating a folder for your language, and your own language support[9]. For reference there are several sample languages to look at. The XMLLanguageSupport.java is a good choise as it is short and fairly easy to comprehend. The main job for the LanguageSupport is to link the languages parser (not the Eclipse parser per se) with the RSTA.

7. When creating the parser-file you need to be sure to import the Eclipse model. Then you need to build the class extending the AbstractParser class and implement the empty methods, for reference you should as mention look at some of the sample languages. Things to remember is to register your file-extension with the *EPackage.Registry* and the *Resource.Factory.Registry* (see figure 9.5). Without this, Java will not be able to load the DSL files for either parsing or compiling.

Listing 9.5: Example of EPackage and Factory registry with ThingML

```
// Register the generated package and the
   XMI Factory
EPackage.Registry.INSTANCE.put(
   ThingmlPackage.eNS_URI, ThingmlPackage.
   eINSTANCE);
Resource.Factory.Registry.INSTANCE.
   getExtensionToFactoryMap()
.put("thingml", new ThingmlResourceFactory
   ());
```

8. Every time a user types something into the textarea[10], a method in the LanguageSupport named parse(RSyntaxTextDocument, String) is called. So inside this method, to start parsing the document that RSTA creates of the written code, it need to be converted to an InputStream that the ResourceSetImpl can load. When loaded there are two main options to run, and first the getErrors()-method need to be called to retrieve any syntax-errors. If there are any errors, the method should throw an ParseException to avoid crashing the loader later.

9. If there are no syntax-errors the next step is to try to find context-errors using the resolveAll(). This will try to simulate run-time of the code. If there are any syntax-errors they can be retrieved by yet again calling the getErrors()-method, which will return a list of Diagnostics objects. This in turn needs to be converted into DefaultParserNotice which is what RSTA uses to underline errors in the RTextArea. Every notice needs to be added to the ParseResult object that the parse()-method returns. As stated earlier, if there are some context errors the

[9] Preferably in a DSL/DSLLanguageSupport.java folder hierarchy. Where DSL is the name of the language

[10] RSTA extends JTextArea in the class RTextArea

parser should throw a ParseException which should be caught inside the parse()-method. And it should create DefaultParseNotice similar to the syntax error, but with the result level as *ParserNotice.ERROR*. After all parsing is done, a *firePropertyChange*-method needs to be called.

These steps are all that are needed to get your DSL up and running with RSTA with syntax highlighting, simple auto complete, and syntax and context error underlining. Next step will be about source tree implementation, and step 10 for source tree, and step 14 to utilize the built in content assist for Eclipse.

9.2.2 Implementation of a source tree

10. Adding the source tree can mainly be done in two ways. Here we will only cover the easiest of the two. In which case you need to create your own parser that will build the tree from scratch using tree nodes that extends SourceTreeNode. A good source tree example is the XML one, as it is small with no extra features.

11. How the tree is built inside the tree building class or method is up to you, as long as the AbstractSoruceTree (explained in the next task) can reach the root of the tree.

12. Before testing the tree builder a DefaultCellTreeRenderer and a AbstractSourceTree need to be built. The initial CellRenderer is there to populate the tree and to avoid using Swing's HTML rendering engine, as it is very slow. Still, there will be problems regarding speed with huge trees. The AbstractSourceTree (which is often named DSLOutlineTree.java) help you manage the tree and textarea interaction. In other words, when to update the tree, when to highlight words marked in the tree and so forth. This class will also retrieve and build a new tree when necessary.

13. When all of these four separated classes have been made, the next step is to integrate the tree with RTextArea, so that it gets generated when a user starts typing. This is done in the refreshSourceTree() method inside the RootPane class.

    This is all that needs to be done to implement a simple source tree for any given language. The only feature not built in RSTA in regards to source trees are the possibility of marking a word in the textarea and at the same time get it selected in the tree.

9.2.3 Adding content assist

Last feature supported by RSTA, is the content assist, or auto-completion as the library is called in RSTA. In RSTA content assist

is done in two ways (which can be mixed). There can be an XML-file with information about your language and keywords (as mention earlier), or by implementing them in code or while the program is running.

14. Let us look at how to do it by code, and how to add the content assist already generated by Eclipse. Some code completion is already implemented in code as seen in the three different *CompletionProvider* method; 'string', 'comment', and 'code'. So if you only need simple completion, these can be added directly here.

15. Adding content assist from the Eclipse model is a little bit more trickier. Since you need to override the keystroke listener inside the inputmap (see an example in listing 9.6). The reason for this override is that RSTA is already listening to the "ctrl+space" hotkey used for content assists. And the only way to insert your own completion dynamically is to hijack the call for completion.

Listing 9.6: Example of how to override Ctrl+Space in RSyntaxTextArea using Inputmap

```
rSyntaxTextArea.getInputMap().put(
   KeyStroke
.getKeyStroke(KeyEvent.VK_SPACE, Event.
   CTRL_MASK), new ContentAssistAction(
   this, autoCodeCompletion));
```

Implementation of the content assist should be done by extending the AbstractAction class. Then the ContentAssistAction need to get a loaded resource from the Eclipse model. A proper implementation of the proposal generated by the Eclipse model can be see seen in listing 9.7 and an explanation can be read in the next section. It is important to remember that the doCompletion() is called last, as this method triggers the content assist window, letting the user see what can be selected.

Listing 9.7: My implementation of the ContentAssistAction

```
static class ContentAssistAction extends
   AbstractAction {

  private AutoCompletion autoCompletion;

  public ContentAssistAction(ThingMLRootPane
     rootPane,
          AutoCompletion autoCompletion) {
 this.autoCompletion = autoCompletion;
 this.rootPane = rootPane;
  }
```

```
 public void actionPerformed(ActionEvent e) {
// Register the generated package and the XMI
   Factory
EPackage.Registry.INSTANCE.put(ThingmlPackage.
   eNS_URI,
          ThingmlPackage.eINSTANCE);
Resource.Factory.Registry.INSTANCE.
   getExtensionToFactoryMap().put(
                "thingml", new
                   ThingmlResourceFactory());

// Load the model
ResourceSet rs = new ResourceSetImpl();
URI xmiuri = URI.createFileURI("Path to
   parseable code");
ThingmlResource nonThingMLresource = rs.
   createResource(xmiuri);
try {
   resource.load(null);
} catch (IOException e) {
   e.printStackTrace();
}

if (resource == null) {// No resources
   available
   autoCompletion.doCompletion();
   return;
}

LanguageAwareCompletionProvider provider = (
   LanguageAwareCompletionProvider)
   autoCompletion
   .getCompletionProvider();
DefaultCompletionProvider dcp = (
   DefaultCompletionProvider) provider
   .getDefaultCompletionProvider();

ThingmlCodeCompletionHelper helper = new
   ThingmlCodeCompletionHelper();
ThingmlCompletionProposal[] proposals = helper
   .computeCompletionProposals(resource,
        rootPane.getCurrentTextArea().getText(),
        rootPane.getCaretPosition());
ThingmlProposalPostProcessor postProcessor =
   new ThingmlProposalPostProcessor();
List<ThingmlCompletionProposal> postProposals =
    postProcessor
   .process(Arrays.asList(proposals));
if (postProposals == null)
   postProposals = java.util.Collections.
      emptyList();

List<ThingmlCompletionProposal> finalProposals
   = new ArrayList<ThingmlCompletionProposal>();
```

```
  for (ThingmlCompletionProposal proposal :
     postProposals) {
     if (proposal.getMatchesPrefix())
    finalProposals.add(proposal);
  }

  dcp.clear();
  for (ThingmlCompletionProposal proposal :
     finalProposals) {
     dcp.addCompletion(new BasicCompletion(dcp,
        proposal
              .getInsertString()));
  }

  autoCompletion.doCompletion();
   }
}
```

An explanation of the Content Assist Action

Inside the actionPerformed() method we first create a resource (in this case a ThingMLResource) for your language using the file that is to be based on for the content assist.

To get the best possible content for the content assist, we use a completionHelper class to compute the first set of completion proposals. Best describe by the comments in the computeCompletionProposals():

> First, we derive all possible proposals from the set of elements that are expected at the cursor position.
>
> Second, the set of left proposals (i.e., the ones before the cursor) is checked for emptiness. If the set is empty, the right proposals (i.e., the ones after the cursor) are also considered. If the set is not empty, the right proposal is discarded, because it does not make sense to propose them until the element before the cursor was completed.
>
> Third, the proposals are sorted according to their relevance. Proposals that match the prefix are preferred over ones that did not. Finally, proposals are sorted alphabetically.

Then we do a postProcess which does nothing, if you have not added any post processing codes earlier in the generation of the code. At the end it just creates all the BasicCompletions that RSTA needs to have to show them in its content assist window.

And again, remember to call doCompletion() when you are done, or else there will be no window popping up. This is because we have written over the original call for completions.

9.3 The new ThingML editor

This building and implementation phase is what led us to the new ThingML Editor as seen in figure 9.4. In the first iteration of development phase only the essential and only a few non-essential features where added. The essential features where the one previously discussed in the chapter “Needs and requirements - The important features”. Except refactoring, which is discussed in the next section, “Challenges”. The two ThingML language essential features where the compiler, a simple project manager.

The non-essential features that where added was tabbing, “format code”, “message field”, samples, and file/text management.

With *tabbing* we mean that when the user opens or create a new file it not open in a new editor but rather in a new text area in a tab behind the current one. This is to mimic how other editors use it, and also to make it easier to work with several files.

The “Format code” feature is a very simple method for indenting the code based on brackets. This was only put in to add some simple cleaning of code that where copy-pasted in under testing.

The idea behind the “message field” was to inform the user about problem that arouse or give hints on different features. In figure 9.4 the message is stating that the user should save the file, to allow the auto-save feature.

When using the *RSTALanguageSupport* as the ground works for the editor, the editor will end up with some non-essential features such at different themes for the editor. Other text editing tools needs to be implemented afterwards, and we added the “copy-paste-cut” combo, and a general file management which lets the user create new file, open file, save, and save as.

Some ThingML samples where also added as a non-essential features. This was donefor two reasons. First, it made it easier to test some of the features as it was a quick way to open small and big projects. And two it would let beginners some easy to reach code/samples to read and test.

9.3.1 Challenges

There are always challenges when building something new. The main challenges was to find out how to properly implement the generated model from EMF. This was mainly because there was little information to be found on how to best interact with the model. A lot of code reading and searching in code was done to just figure out how the “content assists” work and how to retrieve suggest data from it. Other features such as outline, renaming or formatting may be hidden in the model, but has not yet been ‘discovered’.

Something that turned out to be a challenges was how to get the editor to understand how the different ThingML files fits together, since ThingML allows the ‘thing’ and the ‘configuration’ be in separate file. We ended up using a project file called ‘properties’. An example can be seen in listing 9.8 and the editor window is seen in figure 9.5. Since the properties file

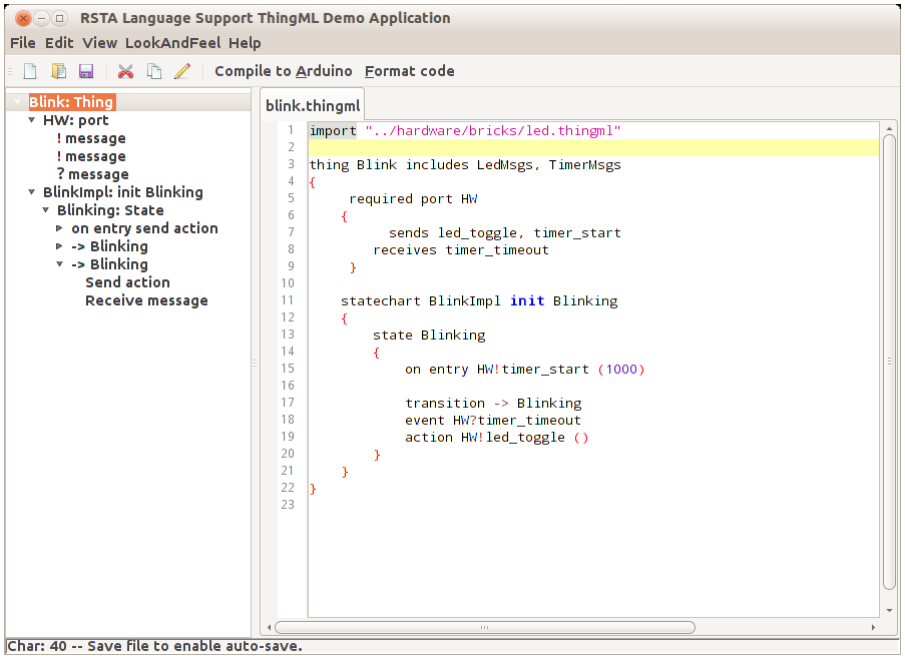


Figure 9.4: The finished ThingML editor built on the RSyntaxTextArea framework

is generated using Java' built in Propteries[11] it automatically adds a time stamp to the file when it as last generated. The actual file includes three keywords; thingml, config, and arduino. For now the 'thingml' keyword is not used. The 'config' keywords hold the address to the 'configuration' for the project, and since only Arduino compiling is implemented for now you can decide which version of Arduino you want to compile to.

Listing 9.8: The properties file for Blink.thingml

```
#Tue Nov 13 11:32:01 CET 2012
thingml=
config=/home/kyrremann/workspace/fork/ThingML/
   org.thingml.editor.rsyntaxtext/src/main/
   resources/samples/samples/\_arduino/blink.
   thingml
arduino=/home/kyrremann/bin/arduino-0022
```

Some of the problems was how to interact with both the ThingML files, the thing and the configuration. This was solved by having a project file for each project. This was also challenging when a user was making a new file, as ThingML need the configuration file to properly compile, and if the user is in the 'thing' file the editor needs to know which configuration file belongs to which file. So one solution is to not allow users to make project

[11] API - java.util.Properties

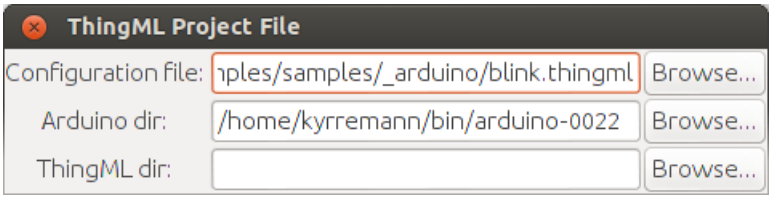


Figure 9.5: Each project has their own properties window

without saving. This is similar to how Eclipse does it with the "new project" wizard.

There are also some challenges with the "message field" area, mostly because it is hidden at the bottom which led to that the test users didn't see it. So important messages should probably be showed in a different fashion.

9.3.2 Future work

There is a lot that has to be done with the editor to get it to a final stage, as this thesis is not focusing of making a finished product. Still there are some features that should have priority.

Refactoring was one of the main features that where supposed to be implemented, unfortunately it proved to complicated to be implemented for the time frame that was set aside for working with the frameworks. This is one of the main features to implement, as it would be interesting to see if it is possible to use EMF for most of the work, or if it needs to be implemented by hand.

Another issue is how the code-base is structured compared to how the original RSytaxTextArea is, as how they are arranged now, updating the RSTA code-base is complicated as lot of the language-specific data is hard coded. Instead of being an extension. When fixed, the advantage would be that updating the RSTA code-base would be a lot simpler and quicker unlike how it is now.

Some of the more easier features to implement are the tools that JSyntaxPane support such as "find and replace" and a project/file tree. The "project tree" could be implemented as a tab underneath the 'outline' to not clutter up the window with to many interactive areas. It would also be convenient to have the configuration added to the main outline. To give an even better overview over the 'thing'.

Lastly there is a need to find out how to best communicate with the developers in the case of errors. The best solution is to have a message pop up and ask the developer for some interaction. As of now there aren't that many use cases where the editor needs feedback from the developer. It is mostly tied to compiling the files.

9.4 Summary

In this chapter we have seen how to implement your own domain-specific language designed in Eclipse Modeling Framework with EMFText, how to export a model which can be implemented with RSyntaxTextArea. The features that where implemented in this chapter are the features that RSyntaxTextArea support out-of-the-box. File handling and other requirements need to be implemented by the developer. Since the project uses Java and Swing, implementing file management is already made available through the Swing API.

Chapter 10

Usability testing of the new ThingML editor

In this chapter the usability testing of the new ThingML editor is presented. It start off by explaining how the testing went ahead and what kind of user base we had gathered. Afterwards we look at what was observed and what kind of feedback we got from the users about the tool. At the end there is a discussion about what can be learned from this usability testing, and if there are some generalisations that can me made.

10.1 The goal for the test study

In this test study we have examined the new ThingML editor to find out;

1. Is the new ThingML editor better then the old editor, and then (of course) why or how?

2. How is it to work with the new ThingML editor?

The first question is aimed as a summative evaluation, where we want to measure how a user think it is to work with the new editor, compared to the old one. In this case we are dependent upon having test users that have used the old ThingML editor.

The second goal of the test study is a more formative evaluation where we wanted to find out how it is as a developer to work with the new editor, and which aspects of the interface are good and bad. Also improvements to the design is something that we are open for.

10.2 The plan for the test study

To find answers to both of the questions asked in the previous section, the test plan issued by Jakob Nielsen in Usability Engineering[13] is used. Not every thing from the list is included since some of them are unnecessary in our test, but the following questions are relevant;

How long is each test session expected to take? We have set aside one hour for the test session. After each session the participant are asked to fill out a questionnaire. The one hour is not included briefing, debriefing and the questionnaire.

What should the state of the system be at the start of the test? To get the most realistic feeling of how it is to work with the editor, the user has to start the software by him/her-self. This is to get a perception of how fast the editor is.

Who are the users going to be? Since we also want a comparison between the two standalone editors we need users that have been working with the old one. This limits the test base to only a handful of users. The two users that have been picked for the usability testing have both been working with ThingML and the old editor, and are or have been a part of the ThingML project at Sintef. It is also important to note that neither of the users have seen the complete editor and how it works.

How many test users are needed? The number of needed test users are hard to establish, but having only two test users is an disadvantage. This is why we want to be careful not to make any broad claims, but keep it generalized.

What test tasks will the users be asked to perform? Each test task will be explained in the next sub section, but in general they focus on problem solving and writing code using the new editor.

What criteria will be used to determine when the users have finished each of the test tasks correctly? It may depend on each of the tasks, but in general we will strive to let accomplishments be the criteria. The user should feel that he or she has accomplished something when they are finished with each task.

What user aids will be made available to the test users? To avoid problems using the ThingML language the website www.thingml.org is available as a helping tool.

What data is going to be collected, and how will it be analyzed once it has been collected? We mainly focused on the interacting with the new editor. Similar to how we did the observation of Eclipse. Thinking out loud was also encouraged. To be more precisely we noted down how the user interact based on the task they are given. After each of the usability testing an analysis of how the user execute each task was conducted. There was also a comparison of both of the users to see if there were some joint problems. Some comparison on how users interact with Eclipse to if some

of the action is replicated in the new editor was also be discussed and analyzed.

What will be the criterion for pronouncing the interface being a success? Since this is only in a prototyping stage, and not aimed at a finished product the interface will not be deemed as a success when the test is over. What we will pronounce is if we feel, based on the data we get, that we are heading in the right direction with the interface. The question should rather be asked what the criterion be for pronouncing the usability testing a success or when we know if we are heading in the right direction. This is also easier to answer. To deem the editor as a success and be able to say that we are heading in the right direction we need feedback that support us in saying that in the same words, that we are heading the right direction. They need to think that this editor is a step further from the original editor, and that we keep true or close to the Eclipse IDE. Even though the editor may be a step further then the original editor, it is important that also the flow is working and that the users can say that the new editor was easier to work with then the old one.

10.2.1 The usability testing tasks

There are 8 task to be performed by the test users, and underneath each of them the goal of the task is listed. The tasks are split into two parts. The first part will focus on learning the tool, featuring step-by-step tasks or tasks asking the user to use a certain feature to complete it. This is to ensure that they user understand at least the basic of using the editor before venturing into the more difficult part. This is where part two start. Part two will feature a more challenging tasks where the user has to make a ThingML project from the ground.

Task zero: Start up ThingML by writing 'ThingML' in the terminal.
Just to get the user going we created a task zero to motivate and give an early feel of accomplishment.

Part one

Task one: Open up the BrokenRobot.thingml. There are, from line 98 to line 110 missing some transitions, plus one syntax error. Fill out the missing transitions and fix the syntax error.
To ease the transition to the new editor we ask the user to start by open an already written ThingML project both of the test users are familiar with. As it's a modified version of the code for a robot, created in a use case in a different ThingML master[15]. The focus of the task will be on the usage of content-assist, and how to use it to find viable variables.

Task two: Navigate using only the outline or the source tree to the state named 'crashed'. This state is missing one transition going to 'forward' with the event for "time_timeout", guard "time == 500", and an action to stop the engine.
This is another task focusing on the features of the editor. Here the user is specifically asked to use only the outline to navigate to the state 'crashed'. When they have navigated to the corresponding state they need to add a new transition based on some guards, events, and actions. Another task is also letting the participant use the content-assist feature. The task was not to focus on how the transition could be, hence all the hints for what the transition was missing.

This task was originally worded to let the user find a syntax error in the code, but because the editor did not generate the outline before all syntax errors where found the task was impossible to complete in the state it was written. Therefore the syntax error was moved up to task one with the missing transitions.

Task three: Close the BrokenRobot.thingml tab.
What will happen when a user closes a tab without saving, and will it be understandable on how they should act? The goal is to see how the user reacts to save action of the editor when closing a tab. Is the action that is asked, the one the user anticipated or expected?

Task four: Open the BrokenRobot.thingml again and the BrokenRobot.thinml config file (can be found inside _arduino). By using the 'Properties' window, tie them together.
To let the user finish the utility testing with a sense of accomplishment, the last task and goal is to let the user learn from what they have done to create and export Brokenrobot.thingml to Arduino.

Task five: Try to export BrokenRobot to Arduino.
When this task was created, we knew that this task would not work, since there is a problem in the BrokenRobot code not working with the new editor. Still we wanted the users to interact with the export button, and also see their reaction when something would not work. We informed both of the users of this problem after they where allowed to try out the task.

Part two

Task six: Open the file "TestCase.thingml".
In the second part the focus is on letting the user interact with the editor just like when they work on their own projects. One problem was that we didn't want them to work with the ThingML configuration file, so we had to create a file with the correct imports first. This is the reason why they have to open a specific file ("TestCase.thingml"), instead of just creating a new one.

Task seven: Make a simple state machine which let the user blink a LED.
To continue on with finding out how users interact with the editor they where asked to write a simple ThingML program which would, when exported to Arduino, blink a LED. This is a kind of "Hello, world!" for ThingML.

Task eight: Compare your Blink state machine with the one in the samples.
This was the last task of the use case, showing the last feature of the editor, 'samples'. The 'sample' list shows examples of easy ThingML programs for novice users to draw inspiration from.

10.3 The questionnaire

After each use case both the participants where asked to answer a set of questions. This was to allow for some reflection of what they had done, and how it was to work with the editor. The focus here was of course the new editor, but also how the participants felt it compared to the two other editors we already have.

0: How did it go working with the new ThingML editor? Keywords; interface, performance. workflow, user experience, functionality.

1: How would you with your own words compare the new ThingML editor with the old editor?

2: If you have used the Eclipse plug-in for ThingML, how would you compare with your own words the new ThingML editor with the Eclipse plug-in?

3: Was there something in particular you did not like with the new ThingML editor?

4: Do you think this is a good application, and would you prefer to use it for your next ThingML project?

10.4 The first use case

The first use case was with a master student also working with the ThingML project. The student had earlier worked with ThingML and was familiar with the old ThingML Editor and with the language. I started with presenting the task at hand and that the participant was to try out the new editor and its features. Thinking out loud was also encouraged. I also mentioned that I would take notes during the study, and that I would present a new task when the participants felt that they where done with the last one and wanted a new one.

As mentioned each task was presented one at a time. In the beginning after receiving task number one some time where spent on the ThingML.org site to brush up on the ThingML knowledge. This was not a problem and even encouraged. When the knowledge was sufficient the participant managed to solve task one at ease. When we switched to task two a problem was discovered. The task was to navigate to a state 'left' only using the outline of the tool, but the outline was empty. It was later discovered this was true because there was a spelling error later in the code, which when fixed produced the outline.

Continue on with task three and four, where task three was easy and understandable, while task four let to some confusion. The task was to use the built in export button for Arduino. When the participant pressed the export button nothing happened. We could see in the log that the process of compiling ThingML code and generating Arduino code was happening, but when logging retrospecting back to the log it seems that there was some connection problems with the ThingML State Machine and the ThingML Configuration code, which ended up telling the compiler that there where no state machine to compile.

The fist part was to teach the user how to use the editor, while the second part focused on how to utilize the editor writing a simple state machine. As task six was just a set-up phase there wasn't much to report from. Task seven on the other hand proved to be a bigger challenge. In this task the participant was asked to create a State Machine with at least one state, which would blink a led using a timer. One of the major challenges was that the participant was not aware of where the files to import were stored, so it was skipped, which later proved to pose challenges. One of the main challenges was the lack of content assists, even though the participant didn't seem to notice since he/she didn't not use it. Another problem was which port to assign, and what messages to send.

10.4.1 Answers from the first user

0: How did it go working with the new ThingML editor? Keywords; interface, performance, workflow, user experience, functionality.
Working with the new ThingML editor was a good experience. I think the interface is well organized, simple and easy to use. It looks similar to other editors, so I easily recognized different items like the menu and the compile

button. The old editor[1] lacked some important and common functionality that is included in the new editor. Two examples are content assist, which both saves you time and helps preventing spelling mistakes, and an outline. It also feels like the performance of the new editor is improved compared to the old one. The old editor sometimes crashed, did not save your project or other unexpected things, and this does not happen with the new editor.

1: How would you with your own words compare the new ThingML editor with the old editor?
With the old editor, getting started with the ThingML language was a bit of work. Having to scroll up and down to find the correct names for ports and transitions takes time, and I sometimes made mistakes that the editor did not notice before compilation. In this way, the editor required a good understanding of the ThingML language before you could use it effectively. The old editor also had some stability problems that sometimes made it crash, not run properly or not save parts of your project.

This problems are not present in the new editor. The new editor also includes an outline that shows all states and each transition from a state. This makes it easier to keep an overview over the project. It checks for bugs and has a content assist, which are important features in an editor.

2: If you have used the Eclipse plug-in for ThingML, how would you compare with your own words the new ThingML editor with the Eclipse plug-in?
I have not used the Eclipse plug-in for ThingML.

3: Was there something in particular you did not like with the new ThingML editor?
I like that you can run the program from the ThingML file, and not the configuration file. I also like the new functionality that was not included in the old editor, in particular the outline and the content assist. These functions makes it faster and easier to develop ThingML programs. I also think it is a nice feature that the user can change the theme of the editor.

I did not like that the outline did not show while the code had a small bug, but this is most likely a mistake that will be fixed.

4: Do you think this is a good application, and would you prefer to use it for your next ThingML project?
Yes, I think this is a good application. Without this improved editor, the ThingML language was harder to use, and I would prefer to use the new editor for my next ThingML project.

10.4.2 Thoughts about the first use case

Some of the major problems using the editor was that there was a lack of consistent feedback from the editor. For example the outline was not

[1]This is an reference to the initial ThingML editor.

always visible because there was an error in the code, and since the editor parse the code and create the outline every time you edit the code, it was often gone. Other problems were to get the content assist working. The content assist is dependent upon having both a ThingML State Machine and a ThingML Configuration to work. This would cause problems when you had just started working with a project, and either didn't have a configuration file or maybe didn't know how to make one[2].

Reading from the answers from the questionnaire, it seems that the new editor is ahead of the older one both in features and performance. Another advantage that the users pointed out is that the new editor looks a lot like other editors or IDEs which the participant think is a huge plus, as it makes it easier to use and navigate in the editor. The outline and content assist which are two of the main features in the new editor are also the two features that get the most attention in the answers given, and are the two features that the participants appreciate the most.

Another problem or lack of a feature was that you had to be in the configuration file when you wanted to export your ThingML project. This was fixed in the new editor, and is also something that was noticed by the participant.

10.5 The second use case

The second use case was with a graduated master student who had already worked with the ThingML language and editor. The use case was changed slightly to correct or avoid some minor challenges that the first version of the tasks. The change was in short to introduce the outline earlier then originally. The use case was held in the same fashion as the first one.

The first part of the use case went rather well with only minor challenges mainly concerning the ThingML language and not the use of the editor. One thing that was notice was the lack of usage or trying to use features built into the editor. There was also a problem with task two. When it got changed to add a new transition instead fixing another error, I forgot to also update the text, so the user was asked to find the wrong 'state'. This let to some confusion since there where no missing transition in the state that the user had found.

Similar to the first case, closing the tab was a small challenge since it was not possible to interact with the tab, but the user quickly found out it was possible to close it under 'edit'. Similar challenge was met when asked to open the properties window and add the configuration file to the ThingML project.

Part two went rather smoothly, but there was no use of the built-in features of the editor.

[2]The lack of content assist would also be a problem if the configuration was in the same file as the state machine. Even thought the ThingML language itself allows it.

10.5.1 Answers from the second user

0: How did it go working with the new ThingML editor? Keywords; interface, performance. workflow, user experience, functionality.
It go'wd[sic] ok. I would like to see keyword highligtning as it was difficult to see if I wrote the keywords correctly, but the sqwiggly line worked fine. Interface: it had the basic functionallity needed from an editor, but I would like to see some kind of autocompletion of keywords and variables. Performance vice, i could write just as fast as in other editors:)

1: How would you with your own words compare the new ThingML editor with the old editor?
The old one had keyword highligning as far as I can remember so that is a plus. But there were problems with the old editor, I just cant remember what it was, but I remeber that it made me frustrated.

2: If you have used the Eclipse plug-in for ThingML, how would you compare with your own words the new ThingML editor with the Eclipse plug-in?
The eclipse plugin have features as code completion and syntax highligtning built in. If we look away from that, the eclipse editor also have a lot of nice functionality from the eclipse framework, such as search, search and replace and other neet functions. But the new editor had a working file/statechart/state/port tree on the left side, which was nice.

3: Was there something in particular you did not like with the new ThingML editor?
hmmmmmmmmm[sic], nothing in particulare comes to mind other than what is already metnioned.

4: Do you think this is a good application, and would you prefer to use it for your next ThingML project? ?
It feels better then the 'native' thingml editor, so it wins over that one, but if it beat the eclipse plugin I don't quite know. It is a little too long since i used it, but I dont think that this editor have a long way to go before it is the best of thoose three.

10.5.2 Thoughts about the second use case

There weren't many new challenges in the second use case, more of a reinforcement of what already was known. What was interesting is the answers from the questions that where asked after the use case. In regards to question 0 the user said that he/she missed the features keyword highlighting and auto-completion. Both of these features are supported by the new editor. The arising of missing two features is then only the fault of me and not making the syntax keyword highlighting clear enough, plus not

having tasks to introduse the user to the content-assist feature, which are both features that the new ThingML editor support.

10.6 Discussion

The utilizing of Jakob Nielsen' test plan really helped in conducting this case study. Both by structuring it in the planning phase and in how it was carried out. This made me as a conductor more prepared, which helped since there were some problems with some of the tasks at hand. Meaning that when the tasks were tested before the first case study there were some obstacles in the sense of missing features in the editor, which means that we couldn't go trough with the first testing as planned and had to postpone both the use cases. Finally when these challenges where fixed, it turned out that task two could not be performed since there was an unknown habit of the parser to not generate the outline if the code would not properly parsed. This was an implementation error, which should not be there, and hence needed to be corrected. Had these surveys been done in the correct order with developing the editor and testing side by side in an agile way, this problem would probably not arise.

With these challenges in the usability testing some of the features went unnoticed. This was made very clear in the second questionnaire. This participant mentioned that both syntax highlighting and auto-completion should have been implemented. Which it was, and something that participant number one mentioned was one of the new features that were nice to have. The second participant also missed features like 'search' and "search and replace" which the Eclipse plug-in has. This was unlike syntax highlighting and auto-completion not implemented in this stage of development, but is something that the RSyntaxTexArea framework supports. Both of the participants appreciated the outline with an overview of the state chart.

One of the reasons why the participants weren't aware of the different features of the editor is because, I as the conductor had thought that the users would experiment more with the editor and then discover the features that where reported missing. In hindsight there should have been a task for each part of the editor to show off what features it supports. One way could have been to let the user do one coding task first without being introduced to the new editor, then a task where the user would learn all the new features, and then finish the use case with a new coding task where the participant could utilize all the new features.

Chapter 11

Conclusion and future work

In this final chapter we address whether or not there is a text-editor framework suitable for implementation of domain-specific languages. In the thesis we have looked into several different ones, and there were more then one good candidate. Still we ended up with RSyntaxTextArea by Fifesoft for our final implementation and testing.

11.1 The thesis question

To try to give a clear answer to the thesis question it has been split into two parts

> Is it possible to make a lightweight editor for a domain-specific language[1] without making it so complex and heavy-weight as an IDE[2], and;

The first part focus on how the final editor should behave, and how the user could interact with it. Both of which is important to have in mind when making a new editor or a new language. Two key factors is the complexity of the editor and the weight of both the downloaded file and how it runs on different computers and operative systems. For comparison we use Eclipse, as it is a very good and versatile IDE with a lot of features. Still there are some drawback with Eclipse that have been discussed in previously chapters, such as complexity and that it is heavy to run.

> Are there frameworks or editors already doing this job and thus answering this question?

The second part opened up for researching other frameworks instead of building a new on from ground. This was done both to support free software and open frameworks, and to save time.

[1] Not necessarily only for a DSL
[2] Such as Eclipse

11.2 Does a framework exists?

Let us start with the second part of the thesis question. In part two several different framework were presented. Some of the framework were chosen because they were known by professors and students, other were chosen by their numbers of features. The four most prominent framework from this informal survey was, Eclipse, JSyntaxPane, jEdit, and RSyntaxTextArea. Eclipse and JSyntaxPane are both used editor for the current ThingML language, but it was desirable to have a further examination of what they were capable of, and to see how they compared to jEdit and RSyntaxTextArea. To find out which framework suited best our presumed needs, we implemented a sub-set of the ThingML language called "Simple ThingML" with each framework. The conclusion of this comparison was that RSyntaxTextArea looked to be the most promising framework. It also helped that RSyntaxTextArea is still under development and that it is a part of an IDE called RText. Both Eclipse and jEdit were too complex and big to be properly utilised. Mostly because both of them does so much more then to help the user program. For example Eclipse features different 'perspectives' when working with different languages, while jEdit was generally difficult to work with and to implement our own language. JSyntaxPane on the other hand turned out, as predicted, to be to scarce.

Based on this comparison there is a positive answer to the second part of the thesis question. In this thesis we have seen that the framework RSyntaxTextArea is a suitable framework. Still there were some challenges when implementing our EMF Ecore model. RSyntaxTextArea is also part of an IDE called RText, but we opted out of using that as the framework to keep the editor as minimal and lightweight as possible. It was also done to avoid a potentially complex or cluttered user interface.

11.3 Implementation of ThingML with RSyntax-TextArea

At the end of this thesis we tried to answer the first part of the thesis question. To see if it was possible to make a lightweight editor with our domain-specific language. We ended using RSyntaxTextArea from Fifesoft as the text-editor framework. In the first iteration most of the features implemented were based on what was believed to be the necessary features for an lightweight text editor. This was later confirmed by both a survey about "features needed in an editor" and the open usage statistics from both Eclipse and Netbeans. Two highly used Java IDEs. After we had implemented ThingML with the new framework there where conducted a usage study of the new editor.

The editor was far from being a finished product, but as stated in the beginning of the thesis we were not aiming for a finished product. Compared to development phases, it would probably be classified as alpha-stage, as most of the features that we want to test out are in the current build. A lot of other useful features and features that users takes for granted

are not implemented in this version. One of the reason for this is that they are not needed to answer the thesis question.

To get a proper test of the new editor we had to use participants already familiar with the language ThingML, so two students with ThingML working experience were chosen. The actual use case of the new tool went rather well, with just some minor bumps in the beginning. The outcome of the two use cases represents one big problems, namely that the features that the editor has was not communicated well enough, and in some cases it wasn't communicated at all. The idea of the implementation of the features were to represent each feature similar to how they are represented in other IDEs, such as Eclipse and Netbeans. This was done by choice to make it easier to transfer to the ThingML editor from other editors. Unfortunately this did not help under the use case. As we can see from the feedback, especially the last user had problems finding and using the features that were implemented. Probably one of the main reason for this challenge was that there were no external testing done under implementation phase of the editor. This could have been solved by using a more test driven software development methodology. This did hurt the process of testing the editor, since a lot of the problems/missing features that were mention in the last use case are all features that the editor supports.

It wasn't all negative feedback from the two use cases, and some of the more positive ones that was reported was that the participants felt that the new editor was easier to use and that it did feel more like an IDE to work with. This makes us confident to say that it is possible to make an lightweight editor for a domain-specific language[3] without making it to complex and heavyweight as an IDE. This is based on what we have learned from the process of finding a framework to implement the ThingML Ecore model with, and from the different surveys, observations, and use cases conducted.

11.4 Future work

Even though the new editor is functional there are still works that needs to be done. Most of the works that needs to be done are to streamline the usage and interactions with the editor. To avoid some of the problems occurred under the use cases. The editor also needs a better project and file manager for creating and opening ongoing ThingML projects. There are also some small noncritical features such as "search and replace" and expand on the tabbed usage in the editor.

[3] In this case ThingML

Bibliography

[1] Tone Bratteteig and Erik Stolterman. 'Design in groups - and all that jazz'. In: *Computers and Design in Contextt* (1997).

[2] Zhixiong Chen and Delia Marx. 'Experiences with Eclipse IDE in programming courses'. In: *J. Comput. Sci. Coll.* 21.2 (Dec. 2005), pp. 104–112. ISSN: 1937-4771. URL: http://dl.acm.org/citation.cfm?id=1089053.1089068.

[3] Steve Easterbrook et al. 'Selecting Empirical Methods for Software Engineering Research'. In: *Guide to Advanced Empirical Software Engineering*. Ed. by Forrest Shull, Janice Singer and Dag I.K. Sjøberg. Springer London, 2008, pp. 285–311. ISBN: 978-1-84800-043-8. DOI: 10.1007/978-1-84800-044-5_11. URL: http://dx.doi.org/10.1007/978-1-84800-044-5_11.

[4] Sven Efftinge and Markus Vülter. *oAW xText: A framework for textual DSLs*. Tech. rep. Ingenieurbüro für Softwaretechnologie, 2006.

[5] Thomas Goldschmidt, Steffen Becker and Axel Uhl. 'Classification of Concrete Textual Syntax Mapping Approaches'. In: *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications* (2008), pp. 169–184.

[6] Hans Grönniger et al. 'Text-based Modeling'. In: *Proceedings of the 4th International Workshop on Software Language Engineering* (Oct. 2007). Nashville, TN, USA, October 2007 Informatik-Bericht Nr. 4/2007, Johannes-Gutenberg-Universität Mainz, October 2007.

[7] Christopher Guntli. *Create a DSL in Eclipse*. Tech. rep. HSR - University of Applied Science in Rapperswil, 2010.

[8] Arthur T Jersild and Margaret F Meigs. 'Direct Observation as a Research Method'. In: *Review of Educational Research* (1939).

[9] Holger Krahn, Bernhard Rumpe and Steven Völkel. 'Efficient Editor Generation for Compositional DSLs in Eclipse'. In: *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM 07)* (2007).

[10] J. Lazar, J.H. Feng and H. Hochheiser. *Research Methods in Human-Computer Interaction*. John Wiley & Sons, 2010. ISBN: 9780470723371. URL: http://www.google.no/books?id=H%5C_r6prUFpc4C.

[11] Bernhard Merkle. 'Textual Modeling Tools: Overview and Comparison of Language Workbenches'. In: *SPLASH'10* (2010).

[12] G.C. Murphy, M. Kersten and L. Findlater. 'How are Java software developers using the Elipse IDE?' In: *Software, IEEE* 23.4 (July 2006), pp. 76–83. ISSN: 0740-7459. DOI: 10.1109/MS.2006.105.

[13] Jakob Nielsen. *Usability engineering*. AP Professional, 1993. ISBN: 0125184069.

[14] Michael Pfeiffer and Josef Pichler. 'A Comparison of Tool Support for Textual Domain-Specific Languages'. In: *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling* (Oct. 2008), pp. 1–7.

[15] Jan Ole Skotterud. 'High level languages on low level devices : Introducing factorized cross-cutting transitions to ThingML'. MA thesis. University of Oslo, 2012.

Chapter 12

Appendix

12.1 Answers for the observation

12.1.1 Participant one

1. I forhold til nivået vi er på nå(INF1010) så vil jeg si middels, kanskje middels+
2. Lastet det ned for lenge siden, men har ikke begynt å bruke det før i det siste.
3. Nei, ingen opplæring. Fått noe råd etc. Jeg hørte om eclipse da jeg fulgte en guide i Java på youtube.
4. Jeg synes det er et bra program. Det gjør en del ting 'enklere', i den grad at det hjelper til med forslag og sparer tid på skriving. Ellers er vel det eneste negative at det noen ganger kan bli litt vanskelig å holde styr på prosjektene, og klassene har en tendens til å havne i et annet prosjekt (kan være enkel nybegynner feil jeg gjør) av og til.

   Ellers er det viktig å ikke gjøre seg for avhengig av eclipse, da det kanskje gjør at man glemmer noen enkle ting som eclipse tar hånd om for deg.

12.1.2 Participant two

1. beginner. Easily not qualified to work with it yet, but with some knowledge of how object oriented programming works, and capable of writing simple programs.
2. 3 weeks
3. Some, but long enough ago to have forgotten most of it. The things I learned did not include shortcuts, commands, utilities etc. but rather how to work with modules, addons and simulators.
4. I like it, and I'm going to continue using it. Allthough it will take some time to get used to the interface and such. In comparison with Sublime (another program) it is both more difficult to navigate and

frustrating to work with for a beginner-programmer, but at the same time it offers a lot of helpful tools witch makes it more comfortable working with when the user begins hoarding methods and classes and needs to recycle old material fast. All in all its a good program, but not really that useful if one is not as patient as me.

12.1.3 Participant three

1. Medium to expert.
2. 3-4 years.
3. No, i google when needed.
4. I like it, gets the job done. I've tried NetBeans too, and I found it slow. I prefer Eclipse of the two.

12.1.4 Participant four

1. medium
2. on and off for 3 months of 2012. reguarly 1 month 2013
3. NO! Except (super good) andoroid eclipse lecture
4. The IDE is good.
   The programming enviroment is hard to understand.
   It is made for experienced users and is not suited for beginners.
   The console is really handy. You dont have to open up a CMD for compiling and running programs.
   The hot key functions in eclipse are good but hard to figure out by your self.
   All in all there are to many functions in eclipse to understand all of them.
   The tool shud'nt have to many options in the begining faces of learning.
   The plugin feature in eclipse gives you acces to any language you want.
   When ever you try to add anything to look sometihng up in eclipse it gives you SOOOOOOOO many options.
   This makes it confusing and hard to figure out what everything is.
   I would wish "a new" text editor based on the IDE would have an advanced option for those who need it, not show it by default.

About your editor needs

This survey is used by Kyrre Havik Eriksen (Kyrrehe@ifi.uio.no) for his master thesis, which is about open-source frameworks for domain-specific language editors. It will take about a minute to complete this survey, and a quick answer would be highly appreciated.

The purpose of this survey is to get an overview of what kind of features different developers use when they write code, and what kind of editors they use.
You may at any time opt-out of this survey, even after you've done it. You will of course remain anonymous.

Thank you for taking the time to answer.

Below is an explanation of the different features used in the survey context:
Syntax highlighting - color highlighting of code/keywords
Code suggestion - gives you information of words reserved to the language (usually in a drop-down menu)
Code completion - similar to code suggestion, but helps you with variable/classes/similar names already written, and from other libraries
Error-reporting - on-the-run parsing of the code which inform you of contextual and syntactical errors
Source tree - A graphical tree representing your code
Refactoring - not in it original form, but helping you rename/move/edit variables/classes/similar in the same file and other associated/referenced files.
*Må fylles ut

How would you rate your developing skills? *

	1	2	3	4	5	6	7	8	9	10	
Just started	○	○	○	○	○	○	○	○	○	○	Experienced programmer

What is your text-editor or IDE of choice? *
Examples; Eclipse, Emacs, Vim, Netbeans, IntelliJ, Gedit, and so on.

- [ ] Eclipse
- [ ] Emacs
- [ ] Vim
- [ ] NetBeans
- [ ] IntelliJ IDEA
- [ ] Visual Studio
- [ ] Gedit/Notepad/similar
- [ ] Xcode
- [ ] Andre:

Does your choice of IDE/text-editor support one or more of the following features? *
See transfiguration/explanation of the features in the beginning of the survey.

- [ ] Syntax highlightning

- [ ] Code suggestion
- [ ] Code completion
- [ ] Error-reporting
- [ ] Source tree
- [ ] Refactoring

Which of the following features do you think is necessary for an IDE/text-editor? *

- [ ] Syntax highlightning
- [ ] Code suggestion
- [ ] Code completion
- [ ] Error-reporting
- [ ] Source tree
- [ ] Refactoring
- [ ] None (please elaborate in the next question)

If you answered "none" in the previous qusetion, you can use this textbox to elaborate.

Which of the following features do you use when you develop? *

	Not used	2	Uses	4	Uses all the time
Syntax highlightning	○	○	○	○	○
Code suggestion	○	○	○	○	○
Code completion	○	○	○	○	○
Error-reporting	○	○	○	○	○
Source tree	○	○	○	○	○
Refactoring	○	○	○	○	○

If you have other features you use, please specify which.
No need to write an comprehensive answere, just let me know what the feature(s) are called or used

for.

Anything more to add?

Either to this survery, or to features needed to create a good IDE/text-editor.

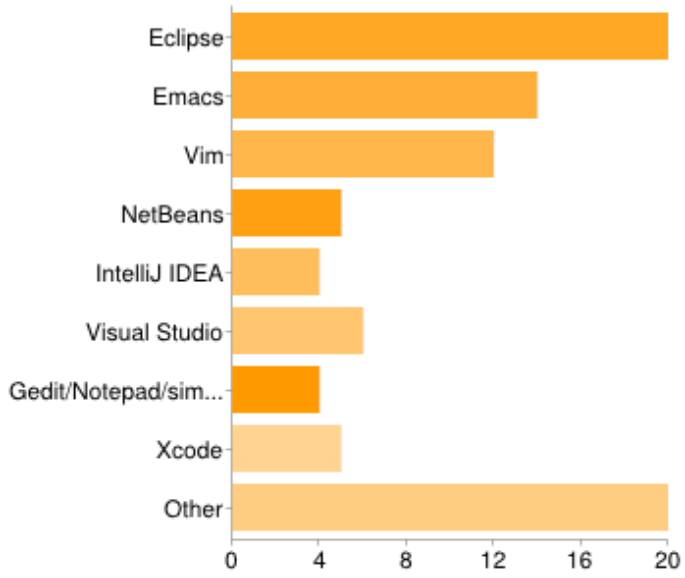
Send

Sammendrag

How would you rate your developing skills?

1 -	Just started	**0**	0%
2		**1**	2%
3		**4**	8%
4		**7**	13%
5		**11**	21%
6		**7**	13%
7		**9**	17%
8		**7**	13%
9		**2**	4%
10 -	Experienced programmer	**4**	8%

What is your text-editor or IDE of choice?



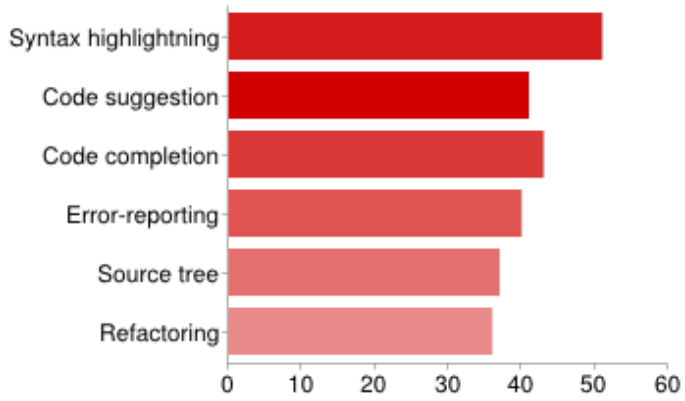
Eclipse	**20**	38%
Emacs	**14**	27%
Vim	**12**	23%
NetBeans	**5**	10%
IntelliJ IDEA	**4**	8%
Visual Studio	**6**	12%
Gedit/Notepad/similar	**4**	8%
Xcode	**5**	10%
Other	**20**	38%

Det er mulig å velge mer enn én avmerkingsboks. Den totale prosenten kan derfor bli mer enn 100 %.

Does your choice of IDE/text-editor support one or more of the following features?

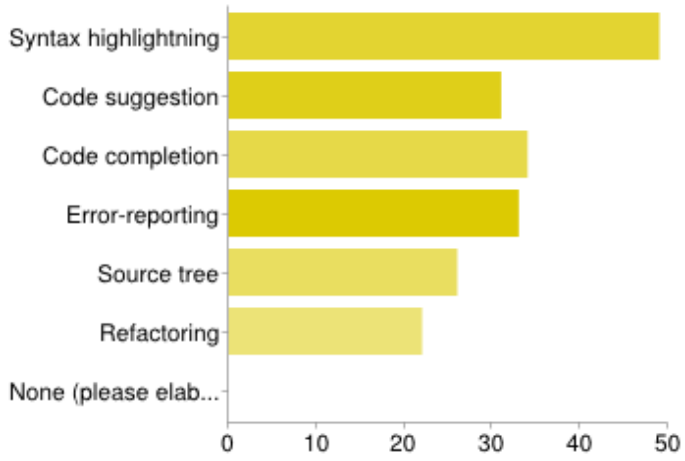
Syntax highlightning	**51**	98%
Code suggestion	**41**	79%
Code completion	**43**	83%
Error-reporting	**40**	77%
Source tree	**37**	71%
Refactoring	**36**	69%

Det er mulig å velge mer enn én avmerkingsboks. Den totale prosenten



kan derfor bli mer enn 100 %.

Which of the following features do you think is necessary for an IDE/text-editor?



Syntax highlightning	**49**	94%
Code suggestion	**31**	60%
Code completion	**34**	65%
Error-reporting	**33**	63%
Source tree	**26**	50%
Refactoring	**22**	42%
None (please elaborate in the next question)	**0**	0%

Det er mulig å velge mer enn én avmerkingsboks. Den totale prosenten kan derfor bli mer enn 100 %.

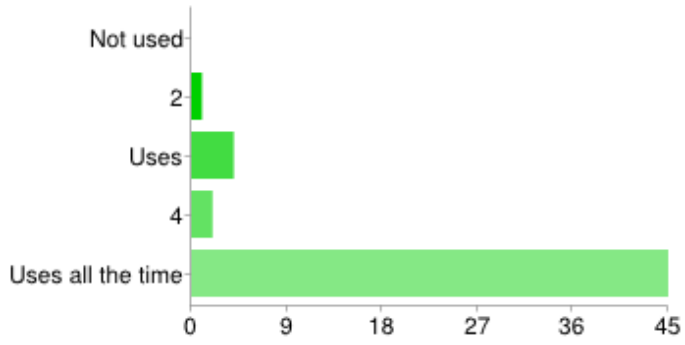
If you answered "none" in the previous qusetion, you can use this textbox to elaborate.

All the rest is very usefull, but Syntax highlighting is the only one I can not be without.

Incremer

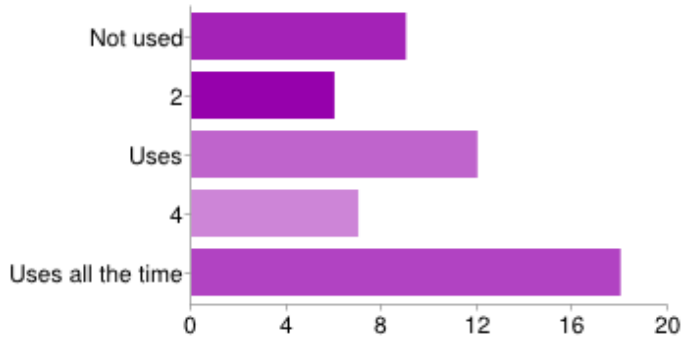
search. Macros.

Which of the following features do you use when you develop? - Syntax highlightning



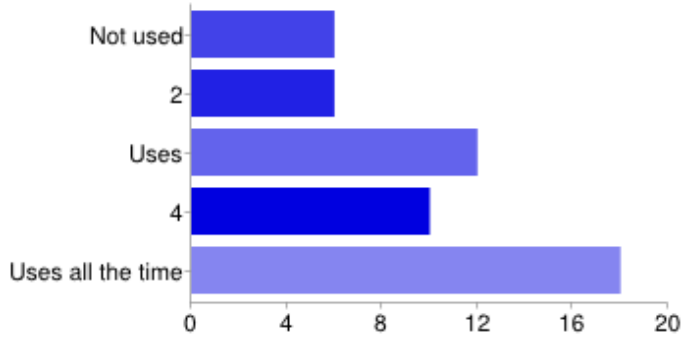
Not used	**0**	0%
2	**1**	2%
Uses	**4**	8%
4	**2**	4%
Uses all the time	**45**	87%

Which of the following features do you use when you develop? - Code suggestion



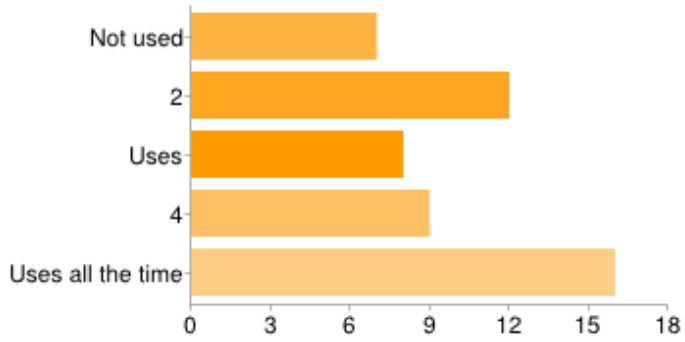
Not used	**9**	17%
2	**6**	12%
Uses	**12**	23%
4	**7**	13%
Uses all the time	**18**	35%

Which of the following features do you use when you develop? - Code completion



Not used	**6**	12%
2	**6**	12%
Uses	**12**	23%
4	**10**	19%
Uses all the time	**18**	35%

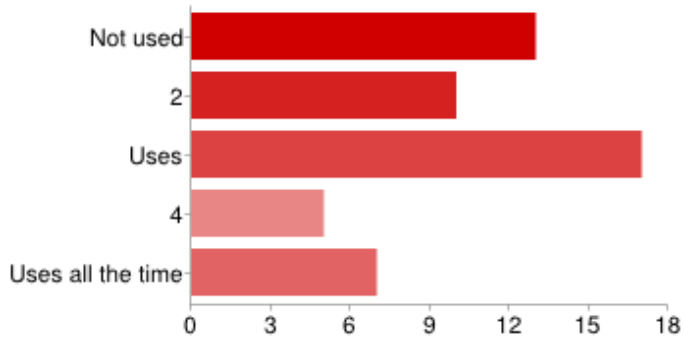
Which of the following features do you use when you develop? - Error-reporting



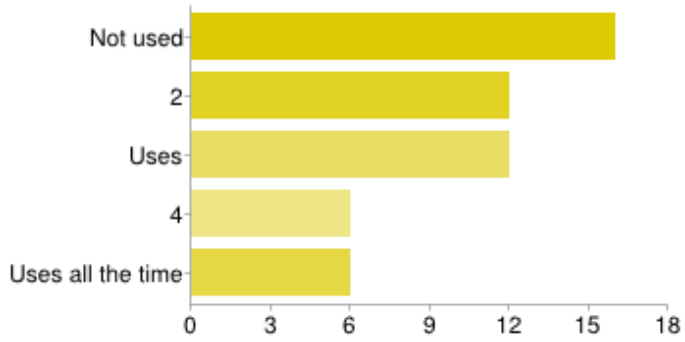
Not used	**7**	13%
2	**12**	23%
Uses	**8**	15%
4	**9**	17%
Uses all the time	**16**	31%

Which of the following features do you use when you develop? - Source tree

Not used	**13**	25%
2	**10**	19%
Uses	**17**	33%
4	**5**	10%
Uses all the time	**7**	13%



Which of the following features do you use when you develop? - Refactoring



Not used	**16**	31%
2	**12**	23%
Uses	**12**	23%
4	**6**	12%
Uses all the time	**6**	12%

If you have other features you use, please specify which.

package manager for easy plugin installation not that I know of Tabbing!!! Javadoc like tooltips(Eclipse) is amazing scripting in elisp, hexeditor interface design, e.g. xcode gui editor Structured editing, integrated REPL, automated documentation, keyboard macros, automated indentation, definition jumping Integrated console, diff view, version control, coverage, dynamic templates, macro recording and key-bound replay, source links (accessing other files via ctrl-click). Keyboard shortcut customization. * clever copy-paste shortcuts (eg Vims cut/copy a hole line in a click) * clever marking of text ...

Anything more to add?

Nope, men sjekk ut epilepsiden.net I have completed the survey based on my HTML/CSS/Javascript experience I love some good shortcuts. Like in emacs and the keybindings I'd like my IDE to be none-obtrusive. I find code completion to be an obtrusive feature most of the time, simply because it disrupts my thinking. It's not necessary a bad thing, I just don't want it available at all times. Being able to turn it off and on (easily) would be a nice feature. A great text-editor *has* to be configurable. I won't use any IDE without vim-keys and editing modes. Intellij also works with DSLs made with MP ...

12.2 ThingML Concrete Syntax Rules

Listing 12.1: The rules used for the concrete syntax for ThingML

```
RULES {
  ThingMLModel::= ( !0 "import" #1 imports[
     STRING_LITERAL] )* ( !0 (types | configs) )*
     ;

  Message ::= "message" #1 name[] "(" (parameters
      ("," #1 parameters)* )? ")"(annotations)* ";
     " ;

  Function ::= "function" #1 name[] "(" (
     parameters ("," #1 parameters)* )? ")"(
     annotations)* ( #1 ":" #1 type[] ( "["
     cardinality "]")? )? #1 body ;

  Thing::= "thing" (#1 fragment[T_ASPECT])? #1
     name[] (#1 "includes" #1 includes[] ("," #1
     includes[])* )? (annotations)* !0 "{" (
     messages | functions | properties | assign |
     ports | behaviour )* !0 "}" ;

  RequiredPort ::= !1 (optional[T_OPTIONAL])? "
     required" #1 "port" #1 name[] (annotations)*
     !0 "{" ( "receives" #1 receives[] ("," #1
     receives[])* | "sends" #1 sends[] ("," #1
     sends[])* )* !0 "}" ;

  ProvidedPort ::= !1 "provided" #1 "port" #1
     name[] (annotations)* !0 "{" ( "receives" #1
     receives[] ("," #1 receives[])* | "sends" #1
     sends[] ("," #1 sends[])* )* !0 "}" ;

  Property::= !1 (changeable[T_READONLY])? "
     property" #1 name[] #1 ":" #1 type[] ( "["
     cardinality "]")? (#1 "=" #1 init)? (
     annotations)*;

  Parameter::= name[] ":" type[] ( "["
     cardinality "]")?;

  PrimitiveType::= "datatype" #1 name[] (
     annotations)* ";" ;

  Enumeration::= "enumeration" #1 name[] (
     annotations)* !0 "{" (literals)* "}" ;

  EnumerationLiteral ::= !1 name[] (annotations)*
      ;

  PlatformAnnotation ::= !1 name[ANNOTATION] #1
     value[STRING_LITERAL] ;
```

```
StateMachine::= !1 "statechart" (#1 name[])? #1
    "init" #1 initial[] ("keeps" #1 history[
   T_HISTORY])? (annotations)* #1 "{" ( !1
   properties )* ( !1 "on" #1 "entry" #1 entry )
   ? ( !1 "on" #1 "exit" #1 exit )? ((!1
   substate) | internal)* (!1 region)* !0 (
   transitions )* "}" ;

State::= "state" #1 name[] (annotations)* #1 "{
   " ( !1 properties )* ( !1 "on" #1 "entry"
   entry )? ( !1 "on" "exit" exit )? ( outgoing
   | internal )* !0 "}" ;

CompositeState::= "composite" #1 "state" #1
   name[] #1 "init" #1 initial[] ("keeps" #1
   history[T_HISTORY])? (annotations)* #1 "{" (
   !1 properties )* ( !1 "on" #1 "entry" #1
   entry )? ( !1 "on" #1 "exit" #1 exit )? (
   outgoing | internal | (!1 substate))* (!1
   region)* !0 "}" ;

ParallelRegion ::= "region" #1 name[] #1 "init"
    #1 initial[] ("keeps" #1 history[T_HISTORY])
   ? (annotations)* #1 "{"(!1 substate)* !0 "}"
   ;

Transition::= !1 "transition" (#1 name[])? #1 "
   ->" #1 target[] (annotations)* ( !1 "event"
   #1 event )* ( !1 "guard" #1 guard)? (!1 "
   action" #1 action)? (!1 "before" #1 before)?
   (!1 "after" #1 after)? ;

Transitions ::= !1 "transitions" ( #1 event ) "
   {" !1 (( #1 "(" guard ")" ) ( #1 name[] )? #1
    "->" #1 target[] ( #1 "!" target[] )* ( !1 "
   action" #1 action )? (!1 "before" #1 before)?
    (!1 "after" #1 after)? )* !1 (( "(" guard ")
   " )? ( #1 name[] )? #1 "->" #1 target[] ( #1
   "!" target[] )* ( !1 "action" #1 action )?
   (!1 "before" #1 before)? (!1 "after" #1 after
   )? )? "}" ( #1 "!" target[] )* ;

InternalTransition ::= !1 "internal" (#1 name
   [])? (annotations)* ( !1 "event" #1 event )*
   ( !1 "guard" #1 guard)? (!1 "action" #1
   action)? ;

ReceiveMessage ::= (name[] #1 ":" #1)? port[] "
   ?" message[] ;

PropertyAssign ::= "set" #1 property[] ("["
   index "]")* #1 "=" #1 init ;
```

```
// ******************************
// * Configurations and Instances
// ******************************
Configuration ::= "configuration" (#1 fragment[
   T_ASPECT])? #1 name[] (annotations)* !0 "{" (
    instances | connectors | configs |
   propassigns )* !0 "}" ;

ConfigInclude ::= "group" #1 name[] #1 ":" #1
   config[] (annotations)* !0 ;

Instance ::= "instance" #1 (name[] #1)? ":" #1
   type[] (annotations)* ; // !0 ( assign )* !0

Connector ::= "connector" #1 (name[] #1)? cli "
   ." required[] "=>" srv "." provided[] (!0
   annotations)*;

ConfigPropertyAssign ::= "set" instance "."
   property[] ("[" index "]")* #1 "=" #1 init;

InstanceRef ::= (config[] ".")* instance[];

// *********************
// * Actions
// *********************
SendAction::= port[] "!" message[] "(" (
   parameters ("," #1 parameters)* )? ")";

VariableAssignment ::= property[] #1 ("[" index
    "]")* "=" #1 expression ;

ActionBlock::= "do" ( !1 actions )* !0 "end" ;

LocalVariable::= !1 (changeable[T_READONLY])? "
   var" #1 name[] #1 ":" #1 type[] ( "["
   cardinality "]")? (#1 "=" #1 init)? (
   annotations)*;

ExternStatement::= statement[STRING_EXT] ("&"
   segments)*;

ConditionalAction ::= "if" #1 "(" #1 condition
   #1 ")" !1 action;

LoopAction ::= "while" #1 "(" #1 condition #1 "
   )" !1 action;

PrintAction ::= "print" #1 msg;

ErrorAction ::= "error" #1 msg;

ReturnAction ::= "return" #1 exp;
```

```
FunctionCallStatement ::= function[] "(" (
   parameters ("," #1 parameters)* )? ")";

// ********************
// * The Expressions
// ********************
@Operator(type="binary_left_associative",
   weight="1", superclass="Expression")
OrExpression ::= lhs #1 "or" #1 rhs;

@Operator(type="binary_left_associative",
   weight="2", superclass="Expression")
AndExpression ::= lhs #1 "and" #1 rhs;


@Operator(type="binary_left_associative",
   weight="3", superclass="Expression")
LowerExpression ::= lhs #1 "<" #1 rhs;

@Operator(type="binary_left_associative",
   weight="3", superclass="Expression")
GreaterExpression ::= lhs #1 ">" #1 rhs;

@Operator(type="binary_left_associative",
   weight="3", superclass="Expression")
EqualsExpression ::= lhs #1 "==" #1 rhs;

@Operator(type="binary_left_associative",
   weight="4", superclass="Expression")
PlusExpression ::= lhs #1 "+" #1 rhs;

@Operator(type="binary_left_associative",
   weight="4", superclass="Expression")
MinusExpression ::= lhs #1 "-" #1 rhs;

@Operator(type="binary_left_associative",
   weight="5", superclass="Expression")
TimesExpression ::= lhs #1 "*" #1 rhs;

@Operator(type="binary_left_associative",
   weight="5", superclass="Expression")
DivExpression ::= lhs #1 "/" #1 rhs;

@Operator(type="binary_right_associative",
   weight="5", superclass="Expression")
ModExpression ::= lhs #1 "%" #1 rhs;

@Operator(type="unary_prefix", weight="6",
   superclass="Expression")
UnaryMinus ::= "-" term;

@Operator(type="unary_prefix", weight="6",
   superclass="Expression")
```

```
  NotExpression ::= "not" #1 term;

  @Operator(type="primitive", weight="8",
     superclass="Expression")
  EventReference ::= msgRef[] "." paramRef[];

  @Operator(type="primitive", weight="8",
     superclass="Expression")
  ExpressionGroup ::= "(" exp ")";

  @Operator(type="primitive", weight="8",
     superclass="Expression")
  PropertyReference ::= property[] ;

  @Operator(type="primitive", weight="8",
     superclass="Expression")
  IntegerLiteral ::= intValue[INTEGER_LITERAL];

  @Operator(type="primitive", weight="8",
     superclass="Expression")
  StringLiteral ::= stringValue[STRING_LITERAL];

  @Operator(type="primitive", weight="8",
     superclass="Expression")
  BooleanLiteral ::= boolValue[BOOLEAN_LITERAL];

  @Operator(type="primitive", weight="8",
     superclass="Expression")
  EnumLiteralRef ::= enum[] ":" literal[];

  @Operator(type="unary_postfix", weight="7",
     superclass="Expression")
  ArrayIndex ::= array "[" index "]";

  @Operator(type="primitive", weight="8",
     superclass="Expression")
  FunctionCallExpression ::= function[] "(" (
     parameters ("," #1 parameters)* )? ")";

  @Operator(type="primitive", weight="8",
     superclass="Expression")
  ExternExpression::= expression[STRING_EXT] ("&"
      segments)*;
}
```

Name	Syntax Highlighting	Code completion	Source tree	Syntax folding	Written in
JSyntaxPane	X				Java
Eclipse & EMFText	X	X	X	X	Java & C++
Red Car Editor	X	X	X	X	Ruby
Notepad++	X			X	C++
Codeblocs	X	X	X	X	C++
JeHep	X		X		Java
Face It	X				Java
Apex Text	X	X			Java
GNU Emacs	X	X		X	C & Emacs LISP
jEdit	X	X	X	X	Java
RSyntaxTextArea	X	X	X	X	Java

Table 12.1: An overview of the tools researched for this thesis